

TRANSPORT ARCHITECTURES FOR AN EVOLVING INTERNET

by

KEITH WINSTEIN

Electrical Engineer, Massachusetts Institute of Technology (2014)

Master of Engineering, Massachusetts Institute of Technology (2005)

Bachelor of Science, Massachusetts Institute of Technology (2004)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

at the


MASSACHUSETTS INSTITUTE OF TECHNOLOGY


June 2014


Copyright 2014 Keith Winstein.

The author hereby grants to MIT permission to reproduce and to distribute publicly paper and electronic copies of this thesis document in whole or in part in any medium now known or hereafter created. This work is licensed under the Creative Commons Attribution 4.0 International License.

To view a copy of this license, please visit <http://creativecommons.org/licenses/by/4.0/>.

Author 
Department of Electrical Engineering and Computer Science
May 21, 2014

Certified by 
Hari Balakrishnan
Fujitsu Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by 
Leslie A. Kolodziej
Professor of Electrical Engineering
Chair, Department Committee on Graduate Students

TRANSPORT ARCHITECTURES FOR AN EVOLVING INTERNET

by

KEITH WINSTEIN

Submitted to the Department of Electrical Engineering and Computer Science
on May 21, 2014, in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

Abstract

In the Internet architecture, transport protocols are the glue between an application’s needs and the network’s abilities. But as the Internet has evolved over the last 30 years, the implicit assumptions of these protocols have held less and less well. This can cause poor performance on newer networks—cellular networks, datacenters—and makes it challenging to roll out networking technologies that break markedly with the past.

Working with collaborators at MIT, I have built two systems that explore an objective-driven, computer-generated approach to protocol design. My thesis is that making protocols a *function* of stated assumptions and objectives can improve application performance and free network technologies to evolve.

Sprout, a transport protocol designed for videoconferencing over cellular networks, uses probabilistic inference to forecast network congestion in advance. On commercial cellular networks, Sprout gives 2-to-4 times the throughput and 7-to-9 times less delay than Skype, Apple Facetime, and Google Hangouts.

This work led to Remy, a tool that programmatically generates protocols for an uncertain multi-agent network. Remy’s computer-generated algorithms can achieve higher performance and greater fairness than some sophisticated human-designed schemes, including ones that put intelligence inside the network.

The Remy tool can then be used to probe the difficulty of the congestion control problem itself—how easy is it to “learn” a network protocol to achieve desired goals, given a necessarily imperfect model of the networks where it ultimately will be deployed? We found weak evidence of a tradeoff between the breadth of the operating range of a computer-generated protocol and its performance, but also that a single computer-generated protocol was able to outperform existing schemes over a thousand-fold range of link rates.

Thesis Supervisor: Hari Balakrishnan

Title: Fujitsu Professor of Computer Science and Engineering

Contents

Previously Published Material	7
Acknowledgments	9
1 Introduction	11
1.1 Summary of results	13
1.2 What computers can teach us about congestion control	14
1.2.1 Algorithms could benefit from collecting rate estimates	14
1.2.2 Windowing and pacing work well together	16
1.2.3 TCP-friendliness carries a benefit, but also a cost	17
1.2.4 Assumptions about link rates and multiplexing are important	18
1.3 Congestion control for all	19
2 Sprout: Controlling Delay with Stochastic Forecasts	21
2.1 Context and challenges	24
2.1.1 Cellular networks	24
2.1.2 Measurement example	25
2.1.3 Challenges	26
2.2 The Sprout algorithm	26
2.2.1 Inferring the rate of a varying Poisson process	27
2.2.2 Evolution of the belief state	30
2.2.3 Making the packet delivery forecast	31
2.2.4 Feedback from the receiver to sender	31
2.2.5 Using the forecast	31
2.3 Experimental testbed	33
2.3.1 Saturator	33
2.3.2 Cellsim	34
2.3.3 SproutTunnel	35
2.4 Evaluation	36
2.4.1 Metrics	36
2.4.2 Comparative performance	37
2.4.3 Benefits of forecasting	37
2.4.4 Comparison with in-network changes	39
2.4.5 Effect of confidence parameter	41
2.4.6 Loss resilience	41

2.4.7	Sprout as a tunnel for competing traffic	41
2.5	Sprout in context: the protocol-design contest	42
2.6	Related work	44
2.7	Limitations	45
3	TCP ex Machina: Computer-Generated Congestion Control	47
3.1	Design overview	49
3.2	Summary of results	49
3.3	Related work	50
3.4	Modeling the congestion-control problem	52
3.4.1	Expressing prior assumptions about the network	53
3.4.2	Traffic model	53
3.4.3	Objective function	53
3.5	Generating a congestion-control algorithm	54
3.5.1	Compactly representing the sender's state	55
3.5.2	RemyCC: Mapping the memory to an action	57
3.5.3	Remy's automated design procedure	57
3.6	Evaluation	59
3.6.1	Simulation setup and metrics	59
3.6.2	Single-bottleneck results	63
3.6.3	Cellular wireless links	67
3.6.4	Differing RTTs	70
3.6.5	Datacenter-like topology	71
3.7	Ratatouille, a tool for understanding RemyCCs	71
4	Using Remy to Measure the Learnability of Congestion Control	73
4.1	Summary of results	74
4.2	Protocol design as a problem of learnability	75
4.3	Experimental setup	76
4.3.1	Training scenarios	76
4.3.2	Objective function	76
4.3.3	Evaluation procedure	77
4.4	Measuring the learnability of congestion control	77
4.4.1	Knowledge of network parameters	78
4.4.2	Structural knowledge	78
4.4.3	Knowledge about other endpoints	81
5	Discussion	85

Previously Published Material

Chapter 2 revises a previous publication [65]: K. Winstein, A. Sivaraman, and H. Balakrishnan. Stochastic Forecasts Achieve High Throughput and Low Delay over Cellular Networks. In *NSDI*, Lombard, Ill., April 2013.

Section 2.5 is adapted from [53]: A. Sivaraman, K. Winstein, P. Varley, S. Das, J. Ma, A. Goyal, J. Batalha, and H. Balakrishnan. Protocol Design Contests. *SIGCOMM Computer Communications Review*, July 2014 (forthcoming).

Chapter 3 revises a previous publication [64]: K. Winstein and H. Balakrishnan. TCP ex Machina: Computer-Generated Congestion Control. In *SIGCOMM*, Hong Kong, China, August 2013.

Chapter 4 is adapted from: A. Sivaraman, K. Winstein, P. Thaker, and H. Balakrishnan. An Experimental Study of the Learnability of Congestion Control. In *SIGCOMM*, Chicago, Ill., August 2014 (forthcoming).

Acknowledgments

In one sense, this thesis work has occupied a third of my life—counting from when I became a graduate student in 2003, and keeping the clock running for my sojourns away as I figured out what to work on when I came back. Arriving at this point would have been impossible without the help of a large number of people.

My advisor, Hari Balakrishnan, has been a tireless counselor, mentor, booster, and co-author. I’m lucky to have had the benefit of his wisdom, his nose for interesting problems, and his tendency to get hooked and excited and to champion any idea that comes from his students.

I’m especially grateful to my frequent co-author and labmate, Anirudh Sivaraman, who taught me how to be a productive collaborator, and without whom most of this work could not have been realized.

Thank you to my thesis committee, of Hari Balakrishnan, Dina Katabi, Scott Shenker, and Leslie Kaelbling, who have reached out across disciplines to support and guide this work.

Just before I headed off to MIT as an undergraduate in 1999, I was advised in strong terms to look up Gerald Jay Sussman when I got there. I didn’t know what I was in for! I have worked with Gerry since I was a freshman and cannot easily express how much I have gained from his mentorship, his knowledge, and his sense of taste.

My informal “backup advisors” on the ninth floor—Nickolai Zeldovich and M. Frans Kaashoek—have been patient through innumerable practice talks and late-night idea-bouncing sessions. Outside the ninth floor, I have been fortunate to have the support and counsel of Victor Bahl, Hal Abelson, Mike Walfish, Jonathan Zittrain, and Anne Hunter.

I owe much to my time in the Wall Street Journal’s late Boston bureau. My editors Gary Putka and Dan Golden, and my colleague Charles Forelle, taught me how to dig deeper and how to tell a story. Gary’s “dun-colored warren of cubicles” represented a journalistic Camelot and was as intellectually challenging as any lab.

Many others helped to develop the ideas in this work, especially Pratiksha Thaker, Chris Amato, Andrew McGregor, Tim Shepard, John Wroclawski, Richard Barry, Bill McCloskey, Josh Mandel, Carrie Niziolek, Damon Wischik, Hariharan Rahul, Garrett Wollman, Juliusz Chroboczek, Chris Lesniewski, Vikash Mansinghka, Eric Jonas, Waseem Daher, Jeff Arnold, Marissa Cheng, and Allie Brownell.

To my labmates and ninth-floor cohabitants—Katrina LaCurts, Raluca Ada Popa, Shuo Deng, Lenin Ravindranath, Ravi Netravali, Peter Iannucci, Tiffany Chen, Amy Ousterhout, Jonathan Perry, Neha Narula, Austin Clements, Emily Stark, Cody

Cutler, Eugene Wu, Dan Ports, Silas Boyd-Wickizer, and Sheila Marian—thank you for your companionship and our time together.

Thank you to the subscribers of MIT Zephyr, who organized themselves at the last minute to proofread this dissertation for me: Roger Ford, Laura Baldwin, Jacob Morzinski, Mark Eichin, Carl “Alex” Alexander, Ben Kaduk, and Richard Tibbetts.

Immeasurable gratitude is due to my mom, Joan Winstein, and my sister Allison, for supporting and putting up with me and giving me a sense of how things ought to be. To all our sadness, my father passed away just after making it to Allison’s graduation recital and just after my return to graduate school in 2011. When I was young, my dad would sometimes bring home his graduate students in physics for dinner, some of whom became surrogate older siblings for a time. Much later, for his retirement, they all wrote about his incredible attention to detail and his sense of right and wrong in constructing tests of Mother Nature—things I have always admired and hope have partly rubbed off on me.

When he died, the university was kind enough to let me ghostwrite his obituary, and so my dad will always be remembered the way I saw him. I’ll add one more thing, by dedicating this dissertation to his memory.

Chapter 1

Introduction

Over the last 25 years, the Internet has transformed from an academic networking experiment into a global utility with billions of users. As part of this transformation, every layer of the Internet “stack” has seen dramatic change.

At the link layer, technologies that did not exist 25 years ago now dominate—including wireless local-area networks (Wi-Fi), cellular networks, datacenter interconnects, and transoceanic links with high delay.

One layer up, at the Internet layer, mobility is now ubiquitous. User devices regularly change their interface IP addresses as they roam from network to network.

At the top layer of the Internet stack—the application layer—none of the dominant applications of today existed 25 years ago, including the World Wide Web and its short-length flows, progressive-download video applications (e.g., YouTube and Netflix), and real-time streaming video (e.g., Skype and Facetime).

All of the Internet has had to grapple with this continuous evolution. How should the protocols of the transport layer, sitting in the middle of the stack between the application and the Internet, adapt to changing applications above them and evolving networks below?

One legitimate answer may be that transport-layer protocols need not adapt. If the network or applications evolve, and a transport protocol no longer works adequately alongside them, we can simply stop using the protocol and design a fresh one that matches the new circumstances.

This approach broadly characterizes the Internet’s extraordinarily successful path

Figure 1-1: The Internet’s four-layer model

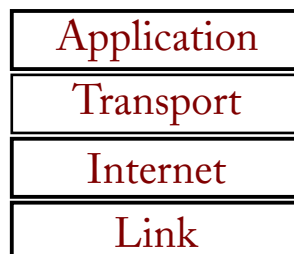
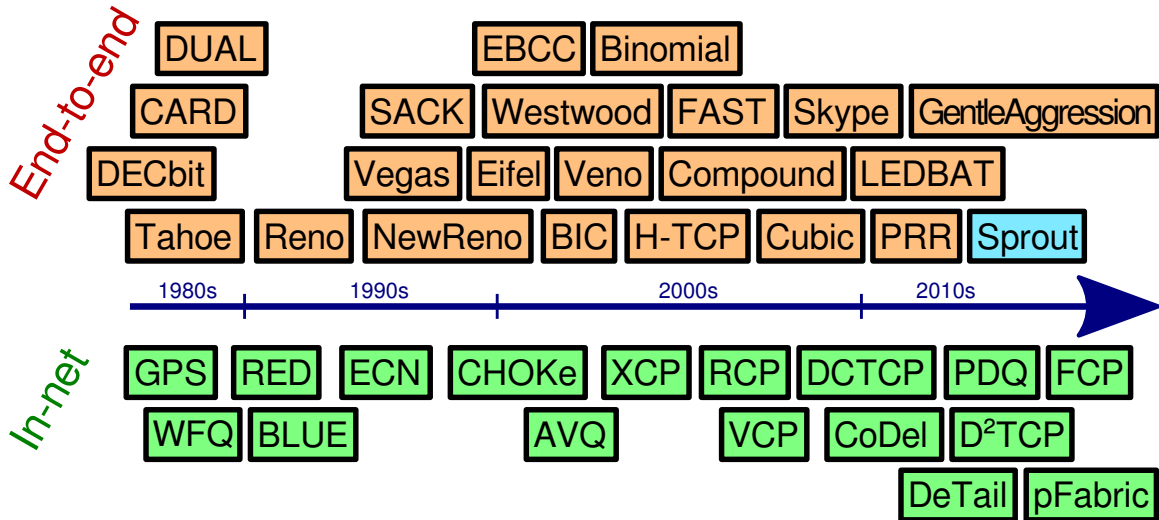


Figure 1-2: As the Internet has evolved, researchers have created at least 40 mechanisms to govern resource allocation on the network—both entirely distributed schemes (“end-to-end”) and ones that include code running “in-net.”



over the last 25 years. Looking at just one function of the transport layer—congestion control, the job of dividing up the network’s resources among contending users—researchers have accommodated new applications and network behaviors by devoting considerable effort to develop new mechanisms, at least 40 in total so far (Figure 1-2).

Despite its success, this approach imposes costs. In defining transport-layer protocols by their *mechanisms*—by the actual behavior of endpoints that execute the protocol—we leave implicit the assumptions that the mechanism makes about the behavior of other layers and the policy that the mechanism is built to pursue.

For contemporary protocols (e.g., TCP Cubic, the current default in Linux), it’s challenging to state the assumptions that the transport makes about lower layers and to predict when those assumptions will no longer hold. This presents a difficulty for link-layer designers who wish to create new networking technologies. Because of Cubic’s prevalence, it has become a *de facto* requirement that Cubic perform well over any network. To achieve adequate performance, the designer of the new link layer must make sure to satisfy Cubic’s implicit assumptions.

This has led to the “bufferbloat” [22] problem: believing that the transport layer will interpret losses as a signal to slow down, the network tries to drop packets as rarely as possible. It has also led to a lack of parallelism in Internet routing—flows only take one path, even if striping across multiple paths could yield a speedup—on the grounds that the transport protocol might interpret out-of-order packets as a pathology.

On the flip side, it is not easy to adjust the transport layer’s assumptions in order to accommodate a new kind of network. There’s no way to tweak the requirements and then retrace the same design process that the protocol’s designers followed, to find the mechanism they would have produced given a different starting point.

This thesis proposes that the transport layer should adapt programmatically to whatever the layers below may do, and whatever the application above it wants. Protocol designers should specify the *policy*—namely, what assumptions they want to make about the network and what kind of performance the application is interested in—and let computers worry about translating this into the mechanism of congestion control.

By “showing our work” clearly enough for a computer to create the design, tweaking a protocol’s assumptions becomes a matter of changing the inputs to a computer program and running it again. My colleagues and I have found that this approach yields better performance than conventional protocols, while allowing adjacent layers to evolve more freely. It’s not clear when computer-generated protocols will be practically deployed on the broad Internet, but what computers can teach us about protocol design can be applied to help humans create better protocols now.

1.1 Summary of results

My colleagues and I have built two systems that explore the benefits of automated, objective-driven congestion-control protocol design.

Sprout (Chapter 2) is a transport protocol designed to carry high-throughput interactive traffic, such as a videoconference, over a cellular network. Sprout includes an explicit model of the dynamics of cellular networks and makes predictions about future performance by inferring a distribution over the current state of the network and evolving the model forward. Its control strategy—how much data to send at a given moment—is a function of those predictions and of an explicit objective: maximize throughput, but cap in-network delays to be at most 100 ms with high probability.

In a trace-driven experimental evaluation (details in §2.4), Sprout gave 2-to-4 times the throughput and 7-to-9 times less delay than Skype, Apple Facetime, and Google Hangouts:

Sprout vs.	Avg. speedup	Delay reduction
Skype	2.2×	7.9×
Hangouts	4.4×	7.2×
Facetime	1.9×	8.7×
Compound	1.3×	4.8×
TCP Vegas	1.1×	2.1×
LEDBAT	no change	2.8×
Cubic	0.91×	79×

Adapted from Figure 2-2.

Remy (Chapter 3) generalizes Sprout to address the classical problem of *multi-agent* congestion control, where independent users contend for the same limited network resource. Remy is a protocol-design tool that takes, as input, a set of assumptions about the uncertain network and workload, and an objective to pursue on behalf of the application.

On a simulated 15 Mbps link with eight senders contending and a round-trip time (RTT) of 150 ms, a computer-generated congestion-control algorithm achieved the following improvements in median throughput and reductions in median queueing delay over these existing protocols:

Protocol	Median speedup	Median delay reduction
Compound	2.1×	2.7×
NewReno	2.6×	2.2×
Cubic	1.7×	3.4×
Vegas	3.1×	1.2×
Cubic/sfqCoDel	1.4×	7.8×
XCP	1.4×	4.3×

Adapted from §3.2.

1.2 What computers can teach us about congestion control

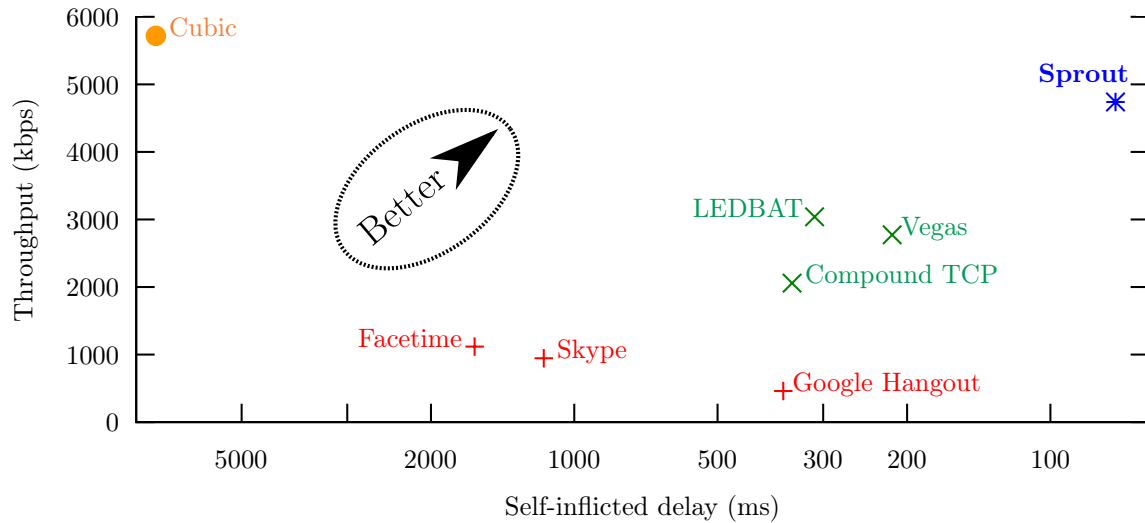
We can learn about transport-protocol design by observing what makes automated protocol-design tools successful. By looking at *what information* turned out to be valuable to Sprout and Remy, and *what behaviors* characterize the successful computer-generated algorithms, we have been able to develop an intuition as to what successful new designs might look like and why Sprout and Remy are able to outperform current protocols.

1.2.1 Algorithms could benefit from collecting rate estimates

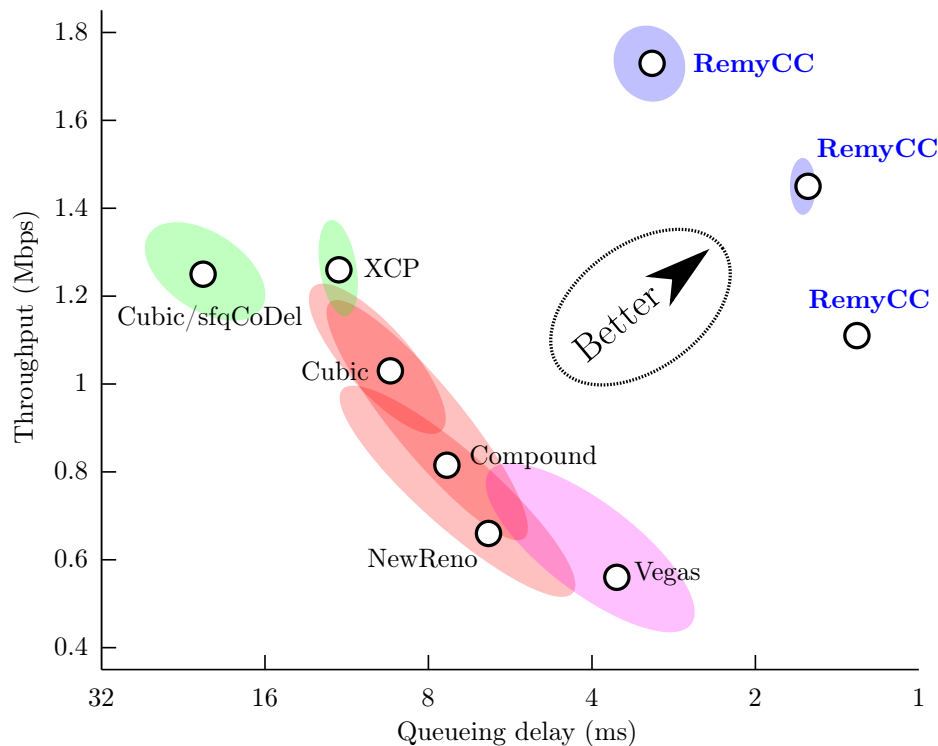
Today’s congestion-control protocols generally pay attention to a small number of signals derived from packets arriving at the receiver, or acknowledgments arriving back at the sender. Classical TCP schemes, such as NewReno [25] and Cubic [23], attempt to detect packet loss as a proxy for in-network buffer overflow. Delay-based schemes such as Vegas [12] detect increasing round-trip delay as a proxy for growing standing queues inside the network.

Researchers and practitioners have proposed that the Internet provide additional information helpful for inferring the presence of congestion [19, 32, 55, 67], but these schemes have not been widely deployed. Sprout and Remy have found that more useful signals can already be gleaned from today’s networks. In particular, estimates of the *rates* of packet transmission and reception proved to be crucial to helping these systems outperform existing algorithms.

Sprout measures the **rate** of packet arrivals at the receiver, forecasts the time required to drain the current queue, and tries to steer this quantity not to exceed a tenth of a second. Based on this technique, Sprout achieves less end-to-end delay than the other algorithms, while achieving close to the throughput of TCP Cubic. From Figure 2-9 (Verizon LTE Downlink):



A Remy-generated algorithm, or RemyCC, measures the **rate** of acknowledgment arrivals, and the **rate** of transmission implied by the echoed sender timestamps in those acknowledgments. We can quantify the value of these signals by removing one of them and rerunning the optimization process. The results are markedly worse with either signal removed. From Figure 3-4:



More broadly, computer-generated protocols have led us to a style of design where the value of information can be evaluated empirically. We may give Remy access to a new congestion signal—either a function of existing TCP acknowledgments from the receiver, or based on a new source of information—and measure how much the resulting congestion-control algorithm improves over the status quo. Similarly, we can knock out an existing signal and rerun Remy to measure how much the loss would hurt performance.

After investigating many candidate signals, we have ended up with computer-generated algorithms that collect four congestion signals (Section 3.5.1):

1. An exponentially-weighted moving average (EWMA) of the interarrival time between new acknowledgments received (`ack_ewma`), where each new sample contributes 1/8 of the weight of the moving average.
2. In some RemyCCs, where the goal is operation over a broad range of networks, we consider the same feature but with a longer-term moving average, using a weight of 1/256 for the newest sample (`slow_ack_ewma`).
3. An exponentially-weighted moving average of the time between TCP sender timestamps reflected in those acknowledgments (`send_ewma`), again with a weight of 1/8 to each new sample.
4. The ratio between the most recent RTT and the minimum RTT seen during the current connection (`rtt_ratio`).

None of these signals is collected by traditional TCP congestion-control schemes. Our simulation experiments indicate each signal is individually valuable on top of the other three.

1.2.2 Windowing and pacing work well together

Traditional congestion-control algorithms are window-based and ack-clocked [54]. Some proposed algorithms use *pacing*—transmitting packets according to an internal clock at the sender—but the performance of these algorithms can display counter-intuitive weaknesses [1].

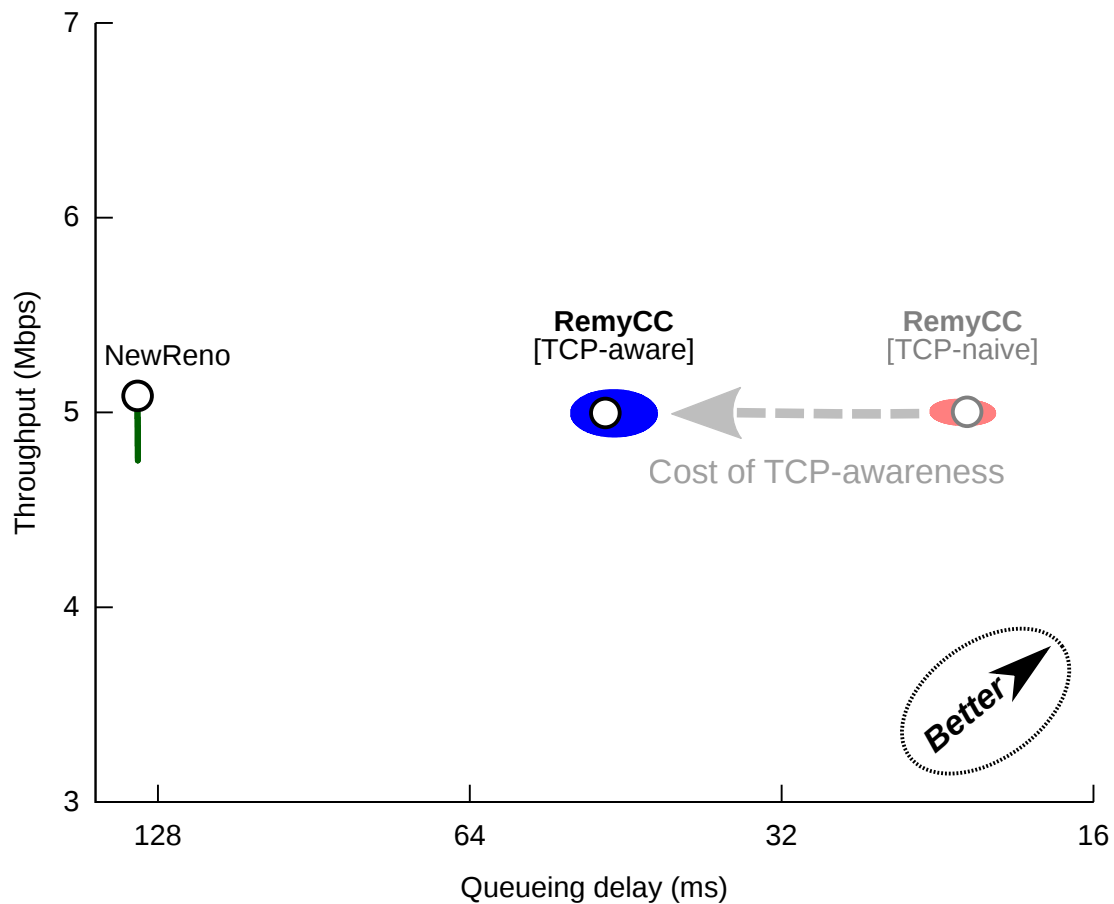
After studying the behavior of Remy’s algorithms and examining their internal logic, we found they often behave as a fine-grained hybrid of these two techniques, qualitatively different in behavior than prior window- or pacing-based algorithms. In steady state, the RemyCCs alternate between windowing and pacing behavior. Within any interval that is at least 2 RTTs, some of the algorithm’s transmissions will have been gated by a windowing constraint (waiting until the congestion window is larger than the number of packets in flight), and some will have been gated by a pacing constraint (waiting until the last packet transmitted was sent long enough ago).

In practice, this hybrid behavior tends to produce quicker convergence to a fair allocation of network resources when a flow starts or finishes than traditional TCP-like schemes. We suspect there may be more to say on the topic of “windowing vs. pacing” if new algorithms similarly attempt to use both techniques nearly concurrently.

1.2.3 TCP-friendliness carries a benefit, but also a cost

Congestion-control protocols are often evaluated for their TCP-friendliness [62], or in other words, for how well they share a contended network resource with traditional TCP flows as another TCP would. By default, RemyCCs do not play well with TCP on the same bottleneck—they are too deferential and end up receiving less than their fair share.¹

This can be corrected, by telling Remy explicitly to assume a nonzero risk that its algorithm may be contending with a traditional TCP flow. In that case, the resulting RemyCCs do manage to claim their fair share of the link (Figure 4-5). But this TCP-friendliness comes at a cost. By making the RemyCC more aggressive in order to match up with TCP, the protocol ends up creating more of a standing queue when it shares the network only with other RemyCCs of the same type. From Figure 4-4:



In other words, TCP-friendliness is a beneficial property *when traditional TCP flows may be present on the same bottleneck*. But it imposes a measurable cost in “clean-slate” designs. On internal networks under the control of one entity, this may be worth taking into account.

¹RemyCCs generally try to keep standing queues small, both because we provide Remy with an objective that penalizes excess delay, but also because the shorter the queueing delay, the quicker the convergence time and the greater the throughput rapidly available to new flows. A long-running traditional TCP flow will work to fill queues until they overflow.

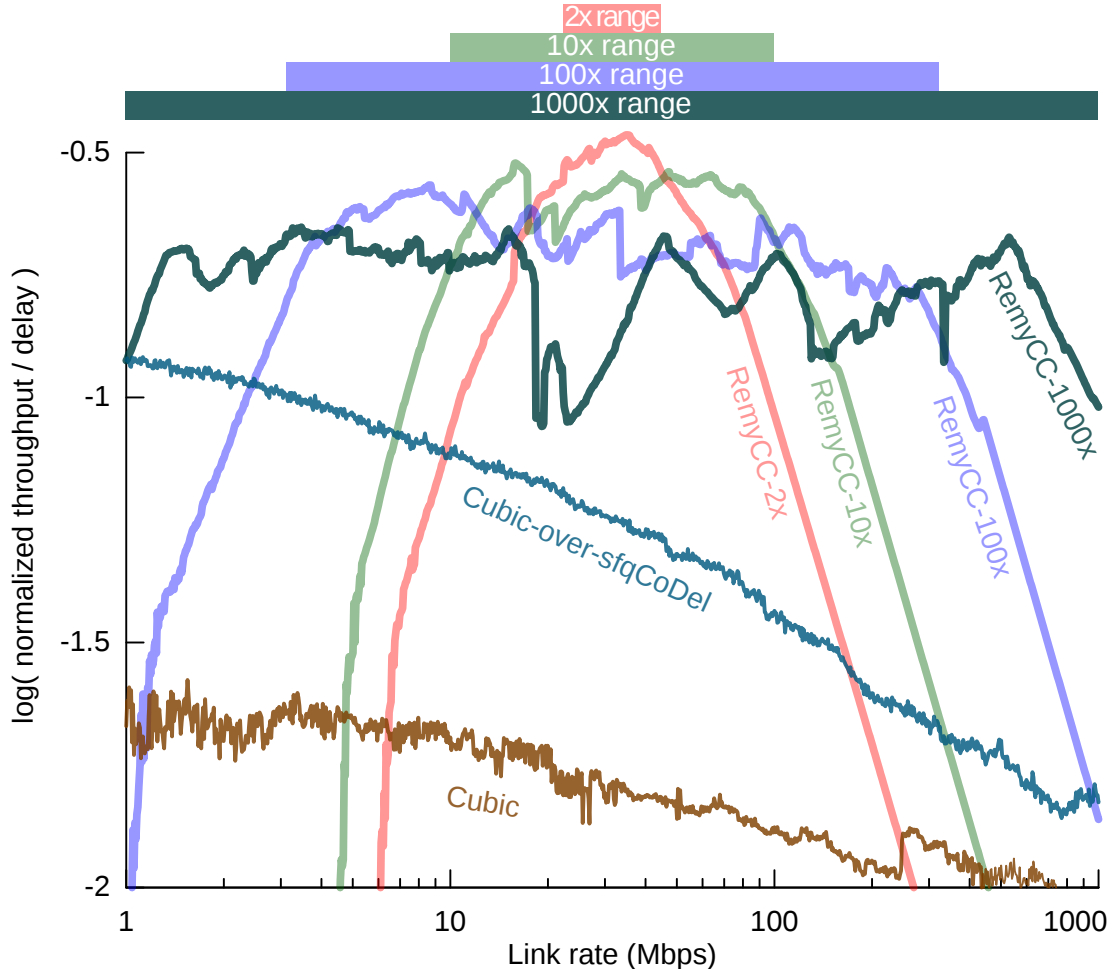
1.2.4 Assumptions about link rates and multiplexing are important

We used Remy as a formalized model of the protocol-design process, to ask rigorously: how faithfully do protocol designers really need to understand the networks they design for?

We found that when modeling the network and link layers, it was important for the designer’s assumptions about *some* network parameters—the bottleneck link rate and the maximum degree of multiplexing at the bottleneck—to include the true values of those parameters.

In one experiment, we asked Remy to generate four different congestion-control protocols, given different amounts of uncertainty in the prior assumptions about the link rate: two-fold uncertainty, ten-fold, hundred-fold, or thousand-fold. We then ran each protocol over a broad range of actual link rates.

We found that performance fell dramatically when the link rate drifted outside an algorithm’s “assumed” range. It was possible for a Remy-generated algorithm to outperform traditional algorithms over the full thousand-fold range—but **only** when this possibility had been included in the algorithm’s prior assumptions. Algorithms with more limited imagination fared poorly. From Figure 4-1:



The results suggest that human-generated protocols are also sensitive to prior assumptions—the performance of Cubic-over-sfqCoDel trailed off at higher link rates, suggesting this scheme may be making an implicit assumption about the bandwidth-delay product that no longer holds at such rates. However, it would be challenging to state explicitly the assumptions of a mechanism like Cubic-over-sfqCoDel.

Among Remy-designed schemes, we also found considerable sensitivity to prior assumptions about the maximum degree of multiplexing at a bottleneck link. By contrast, we found less sensitivity to prior assumptions about the expected distribution of round-trip times or the topology of the network (Section 4.4.2).

1.3 Congestion control for all

Since the 1980s, congestion control has been a well-studied topic in the networking community. Its importance to the Internet may still be growing. Google is putting congestion control in its Chrome Web browser, as part of the QUIC project to send Web traffic over UDP and to allocate limited network resources in a way that optimizes application-specific objectives (e.g., the Web “speed index”).

Netflix, said to account for the majority of Internet traffic in the United States, is working on improving its congestion control in the context of rate selection for progressive-download video, after research findings that its existing algorithm could produce poor results [27].

Meanwhile, “big data” batch processing in specialized datacenters continues to explode. Optimizing the communications of computers connected by a datacenter switching fabric—again, in order to achieve application-specific objectives—has drawn considerable research and commercial effort [3, 4].

I believe that application needs and network behaviors will continue to evolve as the Internet keeps growing and maturing. Congestion control, and the transport layer generally, could adapt gracefully to this innovation if they were merely a *function* of a designer’s network assumptions and application-specific goals. Sprout and Remy represent the first steps of showing what that would look like.

Chapter 2

Sprout: Controlling Delay with Stochastic Forecasts

Cellular wireless networks have become a dominant mode of Internet access. These mobile networks, which include LTE and 3G (UMTS and 1xEV-DO) services, present new challenges for network applications, because they behave differently from wireless LANs and from the Internet’s traditional wired infrastructure.

Cellular wireless networks experience rapidly varying link rates and occasional multi-second outages in one or both directions, especially when the user is mobile. As a result, the time it takes to deliver a network-layer packet may vary significantly, and may include the effects of link-layer retransmissions. Moreover, these networks schedule transmissions after taking channel quality into account, and prefer to have packets waiting to be sent whenever a link is scheduled. They often achieve that goal by maintaining deep packet queues. The effect at the transport layer is that a stream of packets experiences widely varying packet delivery rates, as well as variable, sometimes multi-second, packet delays.

For an interactive application such as a videoconferencing program that requires both high throughput and low delay, these conditions are challenging. If the application sends at too low a rate, it will waste the opportunity for higher-quality service when the link is doing well. But when the application sends too aggressively, it accumulates a queue of packets inside the network waiting to be transmitted across the cellular link, delaying subsequent packets. Such a queue can take several seconds to drain, destroying interactivity (see Figure 2-1).

Our experiments with Microsoft’s Skype, Google’s Hangouts, and Apple’s Facetime running over traces from commercial 3G and LTE networks show the shortcomings of the transport protocols in use and the lack of adaptation required for a good user experience. The transport protocols deal with rate variations in a reactive manner: they attempt to send at a particular rate, and if all goes well, they increase the rate and try again. They are slow to decrease their transmission rate when the link has deteriorated, and as a result they often create a large backlog of queued packets in the network. When that happens, only after several seconds and a user-visible outage do they switch to a lower rate.

This chapter describes Sprout, a transport protocol designed for interactive ap-

Figure 2-1: Skype and Sprout on the emulated Verizon LTE downlink. For Skype, overshoots in throughput lead to large standing queues. Sprout tries to keep each packet's delay less than 100 ms with 95% probability.

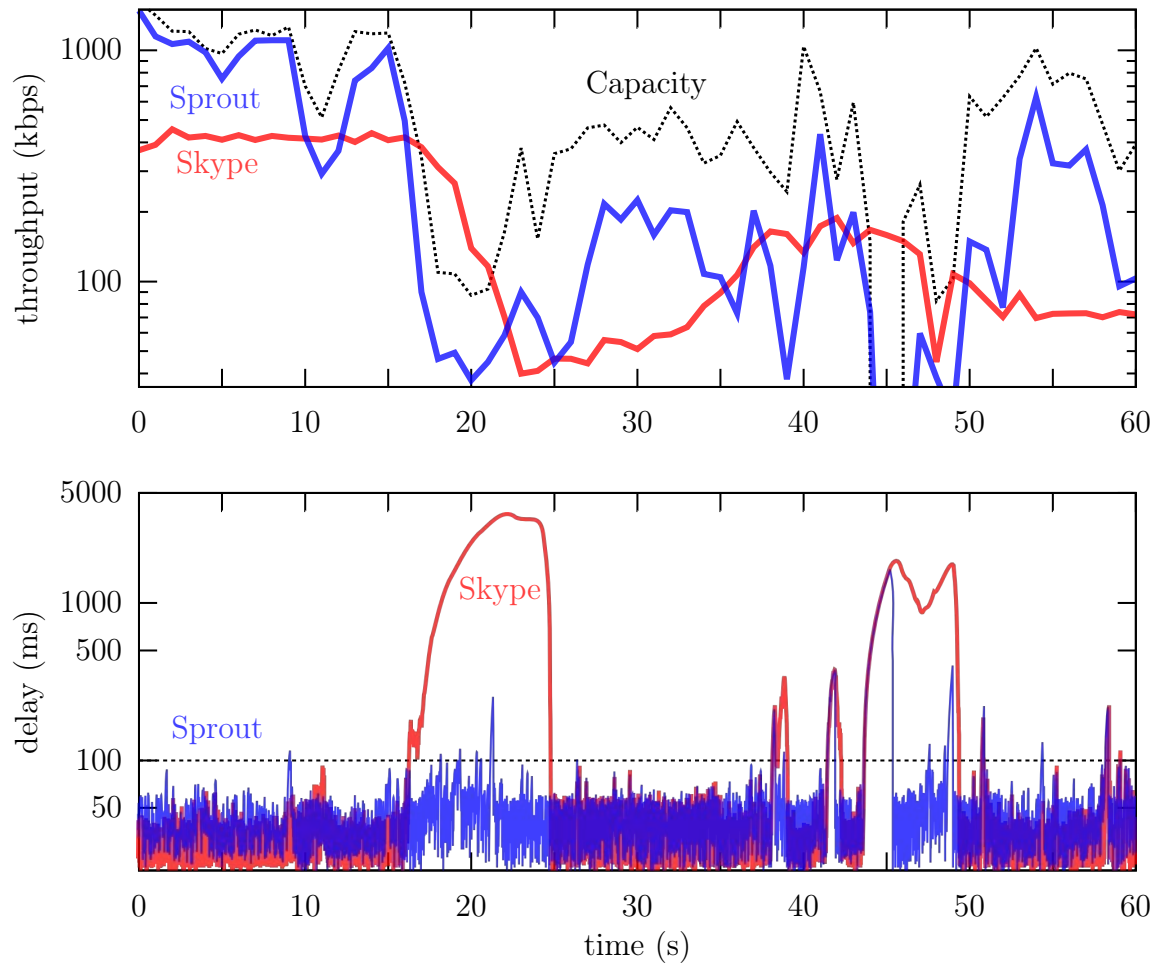


Figure 2-2: Sprout compared with other end-to-end schemes. Measurements where Sprout did not outperform the existing algorithm are highlighted in red.

App/protocol	Avg. speedup vs. Sprout	Delay reduction (from avg. delay)
Sprout	1.0×	1.0× (0.32 s)
Skype	2.2×	7.9× (2.52 s)
Hangouts	4.4×	7.2× (2.28 s)
Facetime	1.9×	8.7× (2.75 s)
Compound	1.3×	4.8× (1.53 s)
TCP Vegas	1.1×	2.1× (0.67 s)
LEDBAT	1.0×	2.8× (0.89 s)
Cubic	0.91×	79× (25 s)
Cubic-CoDel	0.70×	1.6× (0.50 s)

plications on variable-quality networks. Sprout uses the receiver’s observed packet arrival times as the primary signal to determine how the network path is doing, rather than the packet loss, round-trip time, or one-way delay. Moreover, instead of the traditional reactive approach where the sender’s window or rate increases or decreases in response to a congestion signal, the Sprout receiver makes a short-term forecast (at times in the near future) of the bottleneck link rate using probabilistic inference. From this model, the receiver predicts how many bytes are likely to cross the link within several intervals in the near future with at least 95% probability. The sender uses this forecast to transmit its data, bounding the risk that the queuing delay will exceed some threshold, and maximizing the achieved throughput within that constraint.

We conducted a trace-driven experimental evaluation (details in §2.4) using data collected from four different commercial cellular networks (Verizon’s LTE and 3G 1xEV-DO, AT&T’s LTE, and T-Mobile’s 3G UMTS). We compared Sprout with Skype, Hangouts, Facetime, and several TCP congestion-control algorithms, running in both directions (uplink and downlink).

Figure 2-2 summarizes the average relative throughput improvement and reduction in self-inflicted queueing delay¹ for Sprout compared with the various other schemes, averaged over all four cellular networks in both directions.

Cubic-CoDel indicates TCP Cubic running over the CoDel queue-management algorithm [44], which would be implemented in the carrier’s cellular equipment to be deployed on a downlink, and in the baseband modem or radio-interface driver of a cellular phone for an uplink.

We also evaluated a simplified version of Sprout, called Sprout-EWMA, that estimates the network’s future packet-delivery rate with a simple exponentially-weighted

¹This metric expresses a lower bound on the amount of time necessary between a sender’s input and receiver’s output, so that the receiver can reconstruct more than 95% of the input signal. We define the metric more precisely in §2.4.

Figure 2-3: Sprout and Sprout-EWMA achieve comparable results to Cubic assisted by in-network active queue management.

Protocol	Avg. speedup vs. Sprout-EWMA	Delay reduction (from avg. delay)
Sprout-EWMA	1.0×	1.0× (0.53 s)
Sprout	2.0×	0.60× (0.32 s)
Cubic	1.8×	48× (25 s)
Cubic-CoDel	1.3 ×	0.95× (0.50 s)

moving average, rather than using its probabilistic model to make a cautious forecast of future packet deliveries with 95% probability.

Sprout and Sprout-EWMA represents different tradeoffs in their preference for throughput versus delay. As expected, Sprout-EWMA achieved greater throughput, but also greater delay, than Sprout. It outperformed TCP Cubic on both throughput and delay. Despite being end-to-end, Sprout-EWMA outperformed Cubic-over-CoDel on throughput and approached it on delay (Figure 2-3).

We also tested Sprout as a tunnel carrying competing traffic over a cellular network, with queue management performed at the tunnel endpoints based on the receiver's stochastic forecast about future link speeds. We found that Sprout could isolate interactive and bulk flows from one another, dramatically improving the performance of Skype when run at the same time as a TCP Cubic flow.

The source code for Sprout, our wireless network trace capture utility, our trace-based network emulator, and instructions to reproduce the experiments in this chapter are available at <http://alfalfa.mit.edu/>.

2.1 Context and challenges

This section highlights the networking challenges in designing an adaptive transport protocol on cellular wireless networks. We discuss the queueing and scheduling mechanisms used in existing networks, present measurements of throughput and delay to illustrate the problems, and list the challenges.

2.1.1 Cellular networks

At the link layer of a cellular wireless network, each device (user) experiences a different time-varying bit rate because of variations in the wireless channel; these variations are often exacerbated because of mobility. Bit rates are also usually different in the two directions of a link. One direction may experience an outage for a few seconds even when the other is functioning well. Variable link-layer bit rates cause the data rates at the transport layer to vary. In addition, as in other data networks, cross traffic

caused by the arrival and departure of other users and their demands adds to the rate variability.

Most (in fact, all, to our knowledge) deployed cellular wireless networks enqueue each user’s traffic in a separate queue. The base station schedules data transmissions taking both per-user (proportional) fairness and channel quality into consideration [10]. Typically, each user’s device is scheduled for a time slice or resource block in which a variable number of payload bits may be sent, depending on the channel conditions, and users are scheduled in roughly round-robin fashion. The isolation between users’ queues means that the dominant factor in the end-to-end delay experienced by a user’s packets is *self-interaction*, rather than cross traffic. If a user were to combine a high-throughput transfer and a delay-sensitive transfer, the commingling of these packets in the same queue would cause them to experience the same delay distributions. The impact of other users on delay is muted. However, competing demand can affect the throughput that a user receives.

Many cellular networks employ a non-trivial amount of packet buffering. For TCP congestion control with a small degree of statistical multiplexing, a good rule of thumb is that the buffering should not exceed the bandwidth-delay product of the connection. For cellular networks where the “bandwidth” may vary by two orders of magnitude within seconds, this guideline is not particularly useful. A “bufferbloat” [22] base station at one link rate may, within a short amount of time, be under-provisioned when the link rate suddenly increases, leading to extremely high IP-layer packet loss rates (this problem is observed in one provider [40]).

For this reason, the queueing delays in cellular wireless networks cannot simply be blamed on bufferbloat—there is no single buffer size that will produce acceptable results across widely varying link conditions. It is also not simply a question of using an appropriate Active Queue Management (AQM) scheme, because the difficulties in picking appropriate parameters are well-documented and become harder when the available rates change quickly, and such a scheme must be appropriate when applied to all applications, even if they desire bulk throughput. In §2.4, we evaluate CoDel [44], a recent AQM technique, together with a modern TCP variant (Cubic, which is the default in Linux), finding that on more than half of our tested network paths, CoDel slows down a bulk TCP transfer that has the link to itself.

By making changes—when possible—at endpoints instead of inside the network, diverse applications may have more freedom to choose their desired compromise between throughput and delay, compared with an AQM scheme that is applied uniformly to all flows.

Sprout is not a traditional congestion-control scheme, in that its focus is directed at adapting to varying link conditions, not to varying cross traffic that contends for the same bottleneck queue. Its improvements over existing schemes are found when queueing delays are imposed by the user’s own traffic.

2.1.2 Measurement example

In our measurements, we recorded large swings in available throughput on mobile cellular links. Existing interactive transports do not handle these well. Figure 2-1

shows an illustrative section of our trace from the Verizon LTE downlink, whose capacity varied up and down by almost an order of magnitude within one second. From 15 to 25 seconds into the plot, and from 43 to 49 seconds, Skype overshoots the available link capacity, causing large standing queues that persist for several seconds, and leading to glitches or reduced interactivity for the users. By contrast, Sprout works to maximize the available throughput, while limiting the risk that any packet will wait in queue for more than 100 ms (dotted line). It also makes mistakes (e.g., it overshoots at $t = 43$ seconds), but then repairs them.

Network behavior like the above has motivated our development of Sprout and our efforts to deal explicitly with the uncertainty of future link speed variations.

2.1.3 Challenges

A good transport protocol for cellular wireless networks must overcome the following challenges:

1. It must cope with dramatic temporal variations in link rates.
2. It must avoid over-buffering and incurring high delays, but at the same time, if the rate were to increase, avoid under-utilization.
3. It must be able to handle outages without over-buffering, cope with asymmetric outages, and recover gracefully afterwards.

Our experimental results show that previous work (see §2.6) does not address these challenges satisfactorily. These methods are reactive, using packet losses, round-trip delays, and in some cases, one-way delays as the “signal” of how well the network is doing. In contrast, Sprout uses a different signal, the observed arrival times of packets at the receiver, over which it runs an inference procedure to make forecasts of future rates. We find that this method produces a good balance between throughput and delay under a wide range of conditions.

2.2 The Sprout algorithm

Motivated by the varying capacity of cellular networks (as captured in Figure 2-1), we designed Sprout to compromise between two desires: achieving the highest possible throughput, while preventing packets from waiting too long in a network queue.

From the transport layer’s perspective, a cellular network behaves differently from the Internet’s traditional infrastructure in several ways. One is that endpoints can no longer rely on packet drops to learn of unacceptable congestion along a network path ([22]), even after delays reach ten seconds or more. We designed Sprout not to depend on packet drops for information about the available throughput and the fullness of in-network queues.

Another distinguishing feature of cellular links is that users are rarely subject to standing queues accumulated by other users, because a cellular carrier generally provisions a separate uplink and downlink queue for each device in a cell. In a network

where two independent users share access to a queue feeding into a bottleneck link, one user can inflict delays on another. No end-to-end protocol can provide low-delay service when a network queue is already full of somebody else’s packets. But when queueing delays are largely self-inflicted, an end-to-end approach may be possible.

In our measurements, we found that estimating the capacity (by which we mean the maximum possible bit rate or throughput) of cellular links is challenging, because they do not have a directly observable rate *per se*. Even in the middle of the night, when average throughput is high and an LTE device may be completely alone in its cell, packet inter-arrival intervals on a saturated link are highly variable. This is a roadblock for packet-pair techniques ([34]) and other schemes to measure the available throughput.

Figure 2-4 illustrates the *interarrival* distribution of 1.2 million MTU-sized packets received at a stationary cell phone whose downlink was saturated with these packets. For the vast majority of packet arrivals (the 99.99% that come within 20 ms of the previous packet), the distribution fits closely to a memoryless point process, or Poisson process, but with fat tails suggesting the impact of channel-quality-dependent scheduling, the effect of other users, and channel outages, that yield interarrival times between 20 ms and almost four seconds. Such a “switched” Poisson process produces a $1/f$ distribution, or *flicker noise*. The best fit is shown in the plot.²

A Poisson process has an underlying rate λ , which may be estimated by counting the number of bits that arrive in a long interval and dividing by the duration of the interval. In practice, however, the rate of these cellular links varies more rapidly than the averaging interval necessary to achieve an acceptable estimate.

Sprout needs to be able to estimate the link speed, both now and in the future, in order to predict how many packets it is safe to send without risking their waiting in a network queue for too long. An uncertain estimate of future link speed deserves more caution than a precise estimate, so we quantify our uncertainty as well as our best guess.

We therefore treat the problem in two parts. We model the link and estimate its behavior at any given time, preserving the full uncertainty. We then use the model to make forecasts about how many bytes the link will be willing to transmit from its queue in the near future. Most steps in this process can be precalculated at program startup, meaning that CPU usage (even at high throughputs) is less than 5% of a current Intel or AMD PC microprocessor. We have not tested Sprout on a CPU-constrained device or tried to optimize it fully.

2.2.1 Inferring the rate of a varying Poisson process

We model the link as a doubly-stochastic process, in which the underlying λ of the Poisson process itself varies in Brownian motion³ with a noise power of σ (measured

²We can’t say exactly why the distribution should have this shape, but physical processes could produce such a distribution. Cell phones experience fading, or random variation of their channel quality with time, and cell towers attempt to send packets when a phone is at the apex of its channel quality compared with a longer-term average.

³This is a Cox model of packet arrivals [16, 48].

Figure 2-4: Interarrival times on a Verizon LTE downlink, with receiver stationary, fit to a $1/f$ noise distribution.

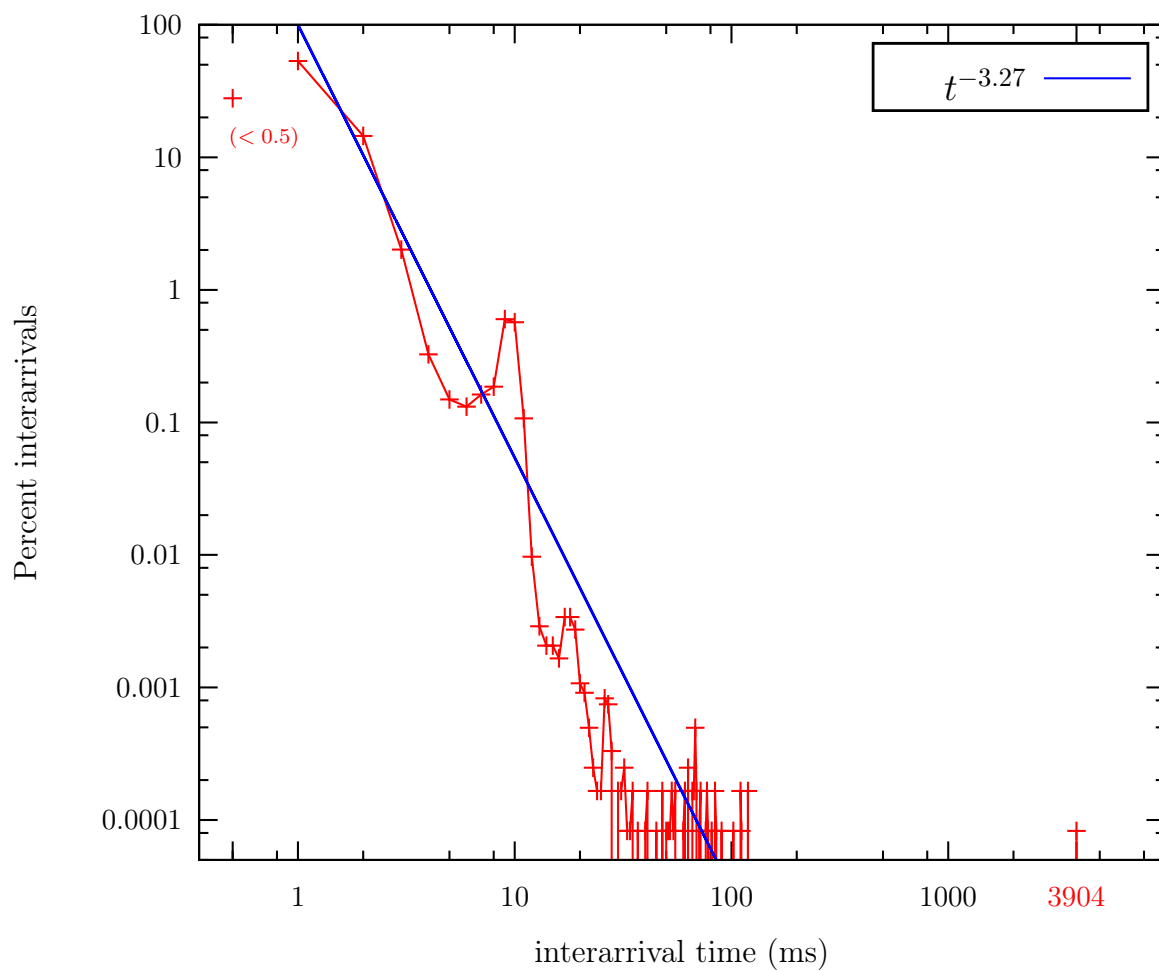
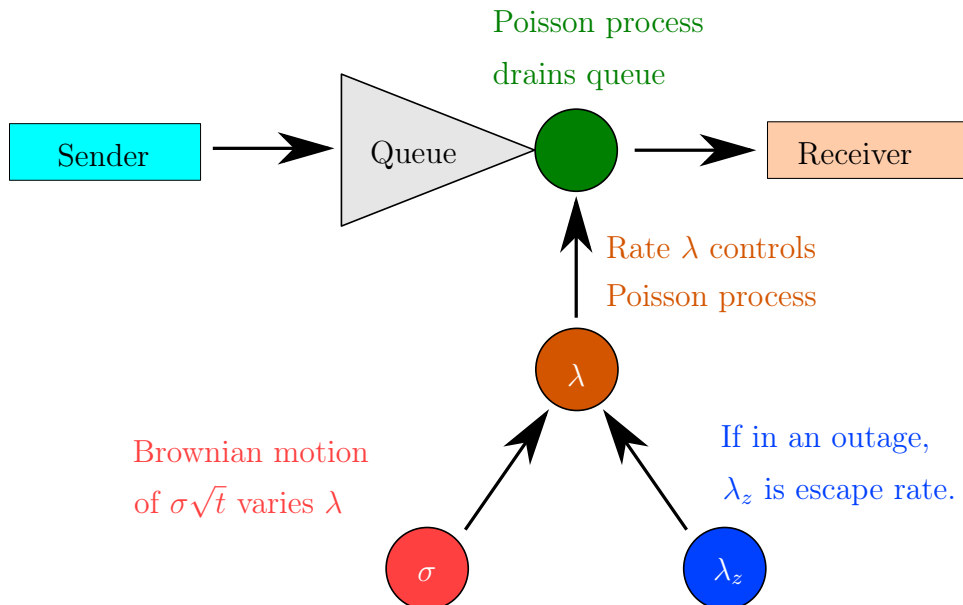


Figure 2-5: Sprout’s model of the network path. A Sprout session maintains this model separately in each direction.



in units of packets per second per $\sqrt{\text{second}}$). In other words, if at time $t = 0$ the value of λ was known to be 137, then when $t = 1$ the probability distribution on λ is a normal distribution with mean 137 and standard deviation σ . The larger the value of σ , the more quickly our knowledge about λ becomes useless and the more cautious we have to be about inferring the link rate based on recent history.

Figure 2-5 illustrates this model. We refer to the Poisson process that dequeues and delivers packets as the service process, or packet-delivery process.

The model has one more behavior: if $\lambda = 0$ (an outage), it tends to stay in an outage. We expect the outage’s duration to follow an exponential distribution $\exp[-\lambda_z]$. We call λ_z the outage escape rate. This serves to match the behavior of real links, which do have “sticky” outages in our experience.

In our implementation of Sprout, σ and λ_z have fixed values that are the same for all runs and all networks. ($\sigma = 200$ MTU-sized packets per second per $\sqrt{\text{second}}$, and $\lambda_z = 1$.) These values were chosen based on preliminary empirical measurements, but the entire Sprout implementation including this model was frozen before we collected our measurement 3G and LTE traces and has not been tweaked to match them.

A more sophisticated system would allow σ and λ_z to vary slowly with time to better match more- or less-variable networks. Currently, the only parameter allowed to change with time, and the only one we need to infer in real time, is λ —the underlying, variable link rate.

To solve this inference problem tractably, Sprout discretizes the space of possible rates, λ , and assumes that:

- λ is one of 256 discrete values sampled uniformly from 0 to 1000 MTU-sized packets per second (11 Mbps; larger than the maximum rate we observed).

- At program startup, all values of λ are equally probable.
- An inference update procedure will run every 20 ms, known as a “tick”. (We pick 20 ms for computational efficiency.)

By assuming an equal time between updates to the probability distribution, Sprout can precompute the normal distribution with standard deviation to match the Brownian motion per tick.

2.2.2 Evolution of the belief state

Sprout maintains the probability distribution on λ in 256 floating-point values summing to unity. At every tick, Sprout does three things:

1. It *evolves* the probability distribution to the current time, by applying Brownian motion to each of the 255 values $\lambda \neq 0$. For $\lambda = 0$, we apply Brownian motion, but also use the outage escape rate to bias the evolution towards remaining at $\lambda = 0$.
2. It *observes* the number of bytes that actually came in during the most recent tick. This step multiplies each probability by the likelihood that a Poisson distribution with the corresponding rate would have produced the observed count during a tick. Suppose the duration of a tick is τ seconds (e.g., $\tau = 0.02$) and k bytes were observed during the tick. Then, Sprout updates the (non-normalized) estimate of the probabilities F :

$$F(x) \leftarrow \Pr_{\text{old}}(\lambda = x) \frac{(x \cdot \tau)^k}{k!} \exp[-x \cdot \tau].$$

3. It *normalizes* the 256 probabilities so that they sum to unity:

$$\Pr_{\text{new}}(\lambda = x) \leftarrow \frac{F(x)}{\sum_i F(i)}.$$

Steps 2 and 3 constitute Bayesian updating of the probability distribution on the current value of λ .

One challenge concerns the situation where the queue is underflowing because the sender has not sent enough. To the receiver, this case is indistinguishable from an outage of the service process, because in either case the receiver doesn’t get any packets.

We use two techniques to solve this problem. First, in each outgoing packet, the sender marks its expected “time-to-next” outgoing packet. For a flight of several packets, the time-to-next will be zero for all but the last packet. When the receiver’s most recently-received packet has a nonzero time-to-next, it skips the “observation” process described above until this timer expires. Thus, this “time-to-next” marking allows the receiver to avoid mistakenly observing that zero packets were deliverable during the most recent tick, when in truth the queue is simply empty.

Second, the sender sends regular heartbeat packets when idle to help the receiver learn that it is not in an outage. Even one tiny packet does much to dispel this ambiguity.

2.2.3 Making the packet delivery forecast

Given a probability distribution on λ , Sprout’s receiver would like to predict how much data it will be safe for the sender to send without risking that packets will be stuck in the queue for too long. No forecast can be absolutely safe, but for typical interactive applications we would like to bound the risk of a packet’s getting queued for longer than the sender’s tolerance to be less than 5%.

To do this, Sprout calculates a *packet delivery forecast*: a cautious estimate, at the 5th percentile, of how many bytes will arrive at its receiver during the next eight ticks, or 160 ms.

It does this by *evolving* the probability distribution forward (without observation) to each of the eight ticks in the forecast. At each tick, Sprout sums over each λ to find the probability distribution of the cumulative number of packets that will have been drained by that point in time. We take the 5th percentile of this distribution as the cautious forecast for each tick. Most of these steps can also be precalculated, so the only work at runtime is to take a weighted sum over each λ .

2.2.4 Feedback from the receiver to sender

The Sprout receiver sends a new forecast to the sender by piggybacking it onto its own outgoing packets.

In addition to the predicted packet deliveries, the forecast also contains a count of the total number of bytes the receiver has received so far in the connection or has written off as lost. This total helps the sender estimate how many bytes are in the queue (by subtracting it from its own count of bytes that have been sent).

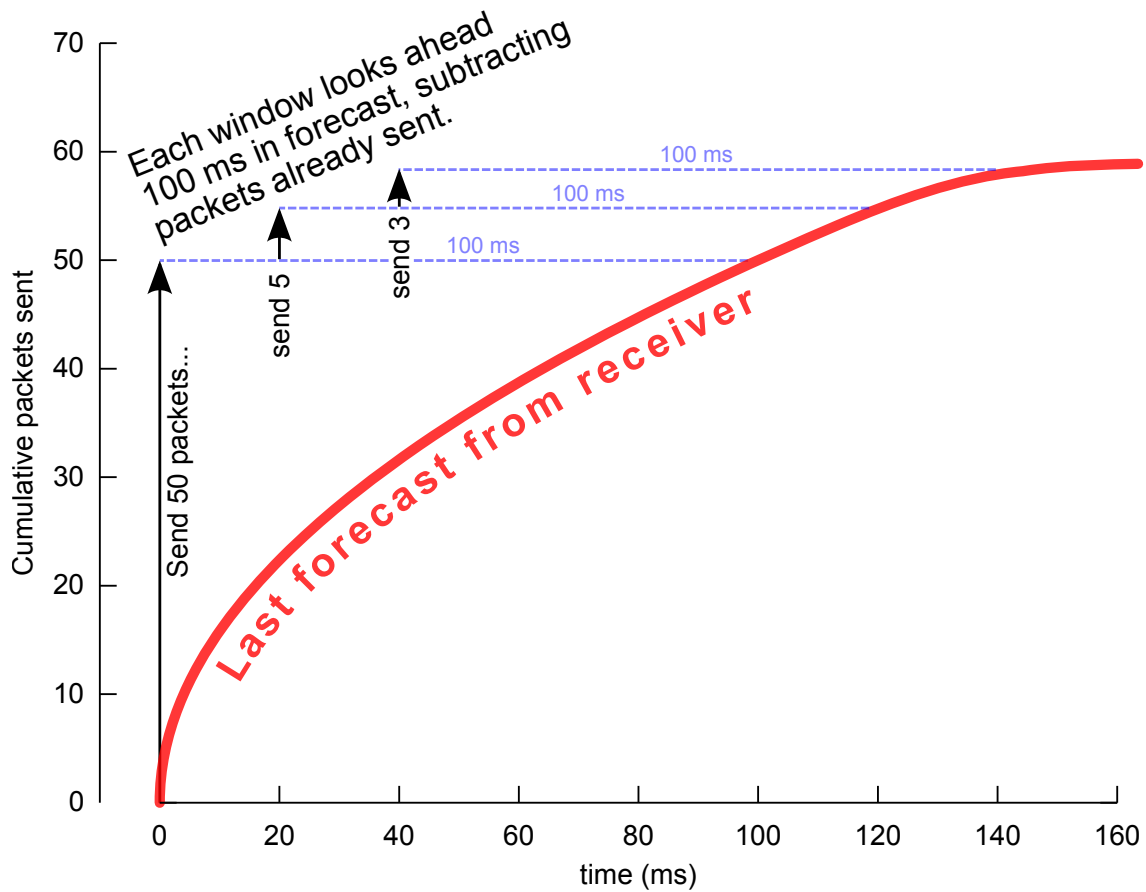
In order to help the receiver calculate this number and detect losses quickly, the sender includes two fields in every outgoing packet: a sequence number that counts the number of bytes sent so far, and a “throwaway number” that specifies the sequence number offset of the *most recent* sent packet that was sent more than 10 ms prior.

The assumption underlying this method is that while the network may reorder packets, it will not reorder two packets that were sent more than 10 ms apart. Thus, once the receiver actually gets a packet from the sender, it can mark all bytes (up to the sequence number of the first packet sent within 10 ms) as received or lost, and only keep track of more recent packets.

2.2.5 Using the forecast

The Sprout sender uses the most recent forecast it has obtained from the receiver to calculate a window size—the number of bytes it may safely transmit, while ensuring that every packet has 95% probability of clearing the queue within 100 ms (a conventional standard for interactivity). Upon receipt of the forecast, the sender timestamps it and

Figure 2-6: Calculating the window sizes from the forecast. The forecast represents the receiver's estimate of a lower bound (with 95% probability) on the cumulative number of packets that will be delivered over time.



estimates the current queue occupancy, based on the difference between the number of bytes it has sent so far and the “received-or-lost” sequence number in the forecast.

The sender maintains its estimate of queue occupancy going forward. For every byte it sends, it increments the estimate. Every time it advances into a new tick of the 8-tick forecast, it decrements the estimate by the amount of the forecast, bounding the estimate below at zero packets.

To calculate a window size that is safe for the application to send, Sprout looks ahead five ticks (100 ms) into the forecast’s future, and counts the number of bytes expected to be drained from the queue over that time. Then it subtracts the current queue occupancy estimate. Anything left over is “safe to send”—bytes that we expect to be cleared from the queue within 100 ms, even taking into account the queue’s current contents. This evolving window size governs how much the application may transmit. Figure 2-6 illustrates this process schematically.

As time passes, the sender may look ahead further and further into the forecast (until it reaches 160 ms), even without receiving an update from the receiver. In this manner, Sprout combines elements of pacing with window-based flow control.

2.3 Experimental testbed

We use trace-driven emulation to evaluate Sprout and compare it with other applications and protocols under reproducible network conditions. Our goal is to capture the variability of cellular networks in our experiments and to evaluate each scheme under the same set of time-varying conditions.

2.3.1 Saturator

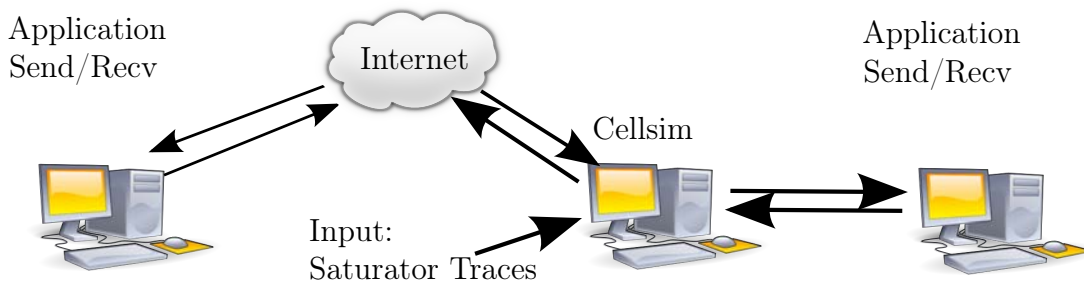
Our strategy is to characterize the behavior of a cellular network by saturating its uplink and downlink at the same time with MTU-sized packets, so that neither queue goes empty. We record the times that packets actually cross the link, and we treat these as the ground truth representing all the times that packets *could* cross the link as long as a sender maintains a backlogged queue.

Because even TCP does not reliably keep highly variable links saturated, we developed our own tool. The Saturator runs on a laptop tethered to a cell phone (which can be used while in a car) and on a server that has a good, low-delay (< 20 ms) Internet path to the cellular carrier.

The sender keeps a window of N packets in flight to the receiver, and adjusts N in order to keep the observed RTT greater than 750 ms (but less than 3000 ms). The theory of operation is that if packets are seeing more than 750 ms of queueing delay, the link is not starving for offered load. (We do not exceed 3000 ms of delay because the cellular network may start throttling or dropping packets.)

There is a challenge in running this system in two directions at once (uplink and downlink), because if both links are backlogged by multiple seconds, feedback arrives too slowly to reliably keep both links saturated. Thus, the Saturator laptop is actually connected to *two* cell phones. One acts as the “device under test,” and its uplink and

Figure 2-7: Block diagram of Cellsim



downlink are saturated. The second cell phone is used only for short feedback packets and is otherwise kept unloaded. In our experiments, the “feedback phone” was on Verizon’s LTE network, which provided satisfactory performance: generally about 20 ms delay back to the server at MIT.

We collected data from four commercial cellular networks: Verizon Wireless’s LTE and 3G (1xEV-DO / eHRPD) services, AT&T’s LTE service, and T-Mobile’s 3G (UMTS) service.⁴ We drove around the greater Boston area at rush hour and in the evening while recording the timings of packet arrivals from each network, gathering about 17 minutes of data from each. Because the traces were collected at different times and places, the measurements cannot be used to compare different commercial services head-to-head.

The Verizon LTE and 1xEV-DO (3G) traces were collected with a Samsung Galaxy Nexus smartphone running Android 4.0. The AT&T trace used a Samsung Galaxy S3 smartphone running Android 4.0, and the T-Mobile trace used a Samsung Nexus S smartphone running Android 4.1.

2.3.2 Cellsim

We then replayed the traces in a network emulator, which we call Cellsim (Figure 2-7). It runs on a PC and takes in packets on two Ethernet interfaces, delays them for a configurable amount of time (the propagation delay), and adds them to the tail of a queue. Cellsim releases packets from the head of the queue to the other network interface according to the same trace that was previously recorded by Saturator. If a scheduled packet delivery occurs while the queue is empty, nothing happens and the opportunity to deliver a packet is wasted.⁵

Empirically, we measure a one-way delay of about 20 ms in each direction on our cellular links (by sending a single packet in one direction on the uplink or downlink back to a desktop with good Internet service). All our experiments are done with this propagation delay, or in other words a 40 ms minimum RTT.

⁴We also attempted a measurement on Sprint’s 3G (1xEV-DO) service, but the results contained several lengthy outages and were not further analyzed.

⁵This accounting is done on a per-byte basis. If the queue contains 15 100-byte packets, they will all be released when the trace records delivery of a single 1500-byte (MTU-sized) packet.

Figure 2-8: Software versions tested

Program	Version	OS	Endpoints
Skype	5.10.0.116	Windows 7	Core i7 PC
Hangouts	as of 9/2012	Windows 7	Core i7 PC
Facetime	2.0 (1070)	OS X 10.8.1	MB Pro (2.3 GHz i7), MB Air (1.8 GHz i5)
TCP Cubic	in Linux 3.2.0	Linux 3.2.0	Core i7 PC
TCP Vegas	in Linux 3.2.0		Core i7 PC
LEDBAT	in μ TP		Core i7 PC
Compound TCP	in Windows 7		Core i7 PC

Cellsim serves as a transparent Ethernet bridge for a Mac or PC under test. A second computer (which runs the other end of the connection) is connected directly to the Internet. Cellsim and the second computer receive their Internet service from the same gigabit Ethernet switch.

We tested the latest (September 2012) real-time implementations of all the applications and protocols (Skype, Facetime, etc.) running on separate late-model Macs or PCs (Figure 2-8).

We also added stochastic packet losses to Cellsim to study Sprout’s loss resilience. Here, Cellsim drops packets from the tail of the queue according to a specified random drop rate. This approach emulates, in a coarse manner, cellular networks that do not have deep packet buffers (e.g., Clearwire, as reported in [40]). Cellsim also includes an optional implementation of CoDel, based on the pseudocode in [44].

2.3.3 SproutTunnel

We implemented a UDP tunnel that uses Sprout to carry arbitrary traffic (e.g. TCP, videoconferencing protocols) across a cellular link between a mobile user and a well-connected host, which acts as a relay for the user’s Internet traffic. SproutTunnel provides each flow with the abstraction of a low-delay connection, without modifying carrier equipment. It does this by separating each flow into its own queue, and filling up the Sprout window in round-robin fashion among the flows that have pending data.

The total queue length of all flows is limited to the receiver’s most recent estimate of the number of packets that can be delivered over the life of the forecast. When the queue lengths exceed this value, the tunnel endpoints drop packets from the head of the longest queue. This algorithm serves as a dynamic traffic-shaping or active-queue-management scheme that adjusts the amount of buffering to the predicted channel conditions.

2.4 Evaluation

This section presents our experimental results obtained using the testbed described in §2.3. We start by motivating and defining the two main metrics: throughput and self-inflicted delay. We then compare Sprout with Skype, Facetime, and Hangouts, focusing on how the different rate control algorithms used by these systems affect the metrics of interest. We compare against the delay-triggered congestion control algorithms TCP Vegas and LEDBAT, as well as the default TCP in Linux, Cubic, which does not use delay as a congestion signal, and Compound TCP, the default in some versions of Windows.

We also evaluate a simplified version of Sprout, called Sprout-EWMA, that eliminates the cautious packet-delivery forecast in favor of an exponentially-weighted moving average of observed throughput. We compare both versions of Sprout with a queue-management technique that must be deployed on network infrastructure. We also measure Sprout’s performance in the presence of packet loss.

Finally, we evaluate the performance of competing flows (TCP Cubic and Skype) running over the Verizon LTE downlink, with and without SproutTunnel.

The implementation of Sprout (including the tuning parameters $\sigma = 200$ and $\lambda_z = 1$) was frozen before collecting the network traces, and has not been tweaked.

2.4.1 Metrics

We are interested in performance metrics appropriate for a real-time interactive application. In our evaluation, we report the *average throughput* achieved and the 95th-percentile *self-inflicted delay* incurred by each protocol, based on measurement of packet entry and exit from the Cellsim.

The *throughput* is the total number of bits received by an application, divided by the duration of the experiment. We use this as a measure of bulk transfer rate.

The *self-inflicted delay* is a lower bound on the end-to-end delay that must be experienced between a sender and receiver, given observed network behavior. We define it as follows: At any point in time, we find the most recently sent packet to have arrived at the receiver. The amount of time since this packet was sent is a lower bound on the instantaneous delay that must exist between the sender’s input and receiver’s output in order to avoid a gap in playback or other glitch at this moment. We calculate this instantaneous delay for each moment in time. The 95th percentile of this function (taken over the entire trace) is the amount of delay that must exist between the input and output so that the receiver can recover 95% of the input signal by the time of playback. We refer to this as “95% end-to-end delay.”

For a given trace, there is a lower limit on the 95% end-to-end delay that can be achieved even by an omniscient protocol: one that sends packets timed to arrive exactly when the network is ready to dequeue and transmit a packet. This protocol will achieve 100% of the available throughput of the link and its packets will never sit in a queue. Even so, the omniscient protocol will have fluctuations in its 95% end-to-end delay, because the link may have delivery outages. If the link does not deliver any

packets for 5 seconds, there must be at least 5 seconds of end-to-end delay to avoid a glitch, no matter how smart the protocol is.⁶

The difference between the 95% end-to-end delay measured for a particular protocol and for an omniscient one is known as the *self-inflicted delay*. This is the appropriate figure to assess a real-time interactive protocol’s ability to compromise between throughput and the delay experienced by users.

To reduce startup effects when measuring the average throughput and self-inflicted delay from an application, we skip the first minute of each application’s run.

2.4.2 Comparative performance

Figure 2-9 presents the results of our trace-driven experiments for each transport protocol. The figure shows eight charts, one for each of the four measured networks, and for each data transfer direction (downlink and uplink). On each chart, we plot one point per application or protocol, corresponding to its measured throughput and self-inflicted delay combination. For interactive applications, high throughput and low delay (up and to the right) are the desirable properties. The table in the introduction shows the average of these results, taken over all the measured networks and directions, in terms of the average relative throughput gain and delay reduction achieved by Sprout.

We found that Sprout had the lowest, or close to the lowest, delay across each of the eight links. On average delay, Sprout was lower than every other protocol. On average throughput, Sprout outperformed every other protocol except for Sprout-EWMA and TCP Cubic.

We also observe that Skype, Facetime, and Google Hangouts all have lower throughput and higher delay than the TCP congestion-control algorithms. We believe this is because they do not react to rate increases and decreases quickly enough, perhaps because they are unable to change the encoding rapidly, or unwilling for perceptual reasons.⁷ By continuing to send when the network has dramatically slowed, these programs induce high delays that destroy interactivity.

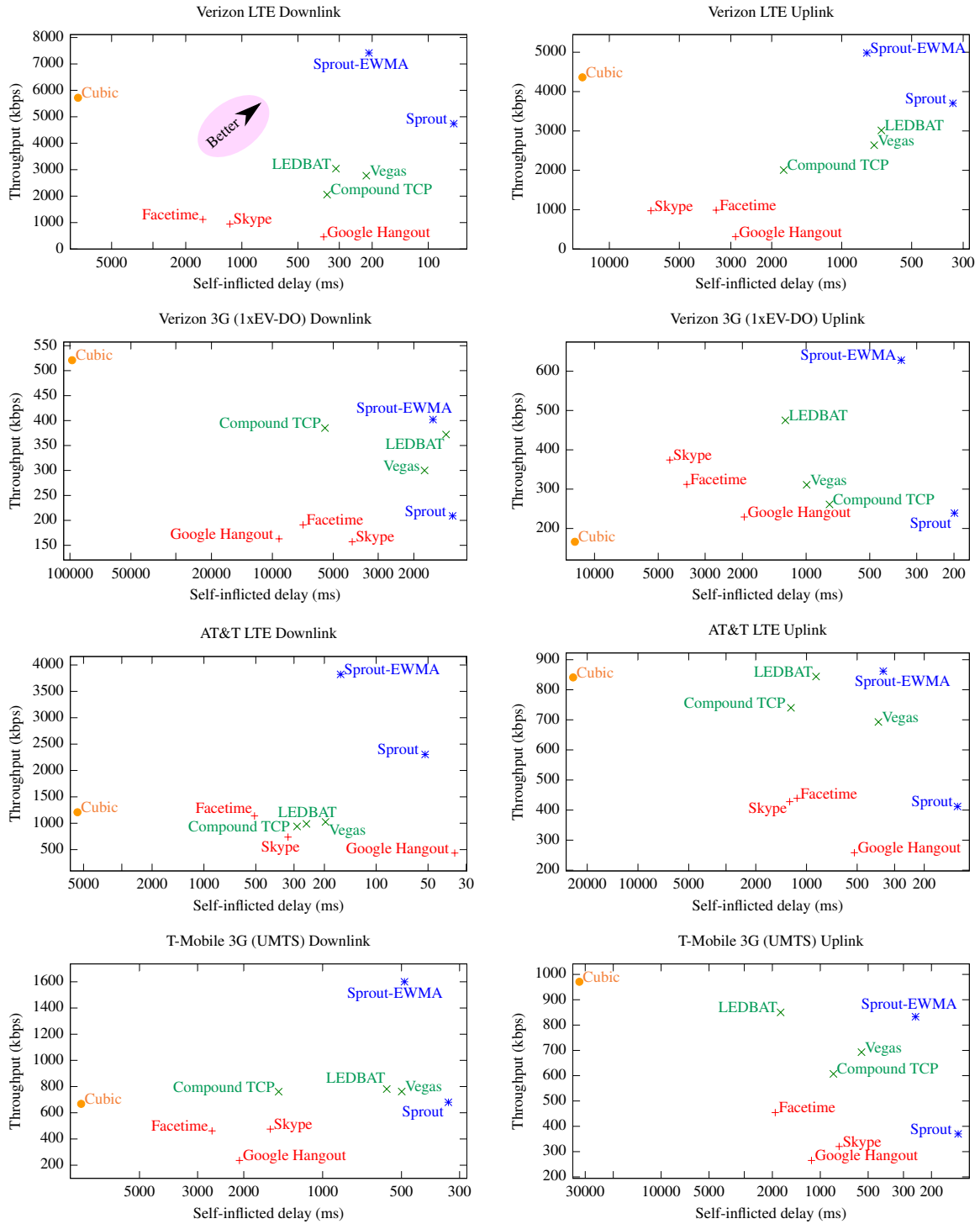
2.4.3 Benefits of forecasting

Sprout differs from the other approaches in two significant ways: first, it uses the packet arrival process at the receiver as the “signal” for its control algorithm (as opposed to one-way delays as in LEDBAT or packet losses or round-trip delays in other protocols), and second, it models the arrivals as a flicker-noise process to perform Bayesian inference on the underlying rate. A natural question that arises is what the benefits of Sprout’s forecasting are. To answer this question, we developed a simple

⁶If packets are not reordered by the network, the definition becomes simpler. At each instant that a packet arrives, the end-to-end delay is equal to the delay experienced by that packet. Starting from this value, the end-to-end delay increases linearly at a rate of 1 s/s, until the next packet arrives. The 95th percentile of this function is the 95% end-to-end delay.

⁷We found that the complexity of the video signal did not seem to affect these programs’ transmitted throughputs. On fast network paths, Skype uses up to 5 Mbps even when the image is static.

Figure 2-9: Throughput and delay of each protocol over the traced cellular links. Better results are up and to the right.



variant of Sprout, which we call *Sprout-EWMA*. Sprout-EWMA uses the packet arrival times, but rather than do any inference with that data, simply passes them through an exponentially-weighted moving average (EWMA) to produce an evolving smoothed rate estimate. Instead of a cautious “95%-certain” forecast, Sprout-EWMA simply predicts that the link will continue at that speed for the next eight ticks. The rest of the protocol is the same as Sprout.

The Sprout-EWMA results in the eight charts in Figure 2-9 show how this protocol performs. First, it out-performs all the methods in throughput, including recent TCPs such as Compound TCP and Cubic. These results also highlight the role of cautious forecasting: the self-inflicted delay is significantly lower for Sprout compared with Sprout-EWMA. TCP Vegas also achieves lower delay on average than Sprout-EWMA. The reason is that an EWMA is a low-pass filter, which does not immediately respond to sudden rate reductions or outages (the tails seen in Figure 2-4). Though these occur with low probability, when they do occur, queues build up and take a significant amount of time to dissipate. Sprout’s forecasts provide a conservative trade-off between throughput and delay: keeping delays low, but missing legitimate opportunities to send packets, preferring to avoid the risk of filling up queues. Because the resulting throughput is relatively high, we believe it is a good choice for interactive applications. An application that is interested only in high throughput with less of an emphasis on low delay may prefer Sprout-EWMA.

2.4.4 Comparison with in-network changes

We compared Sprout’s end-to-end inference approach against an in-network deployment of active queue management. We added the CoDel AQM algorithm [44] to Cellsim’s uplink and downlink queue, to simulate what would happen if a cellular carrier installed this algorithm inside its base stations and in the baseband modems or radio-interface drivers on a cellular phone.

The average results are shown in Figure 2-10. Averaged across the eight cellular links, CoDel dramatically reduces the delay incurred by Cubic, at little cost to throughput.

Although it is purely end-to-end, Sprout’s delays are even lower than Cubic-over-CoDel. However, this comes at a cost to throughput. (Numerical results are given in Figure 2-3.) Sprout-EWMA achieves within 6% of the same delay as Cubic-over-CoDel, with 30% more throughput.

Rather than embed a single throughput-delay tradeoff into the network (e.g. by installing CoDel on carrier infrastructure), we believe it makes architectural sense to provide endpoints and applications with such control when possible. Users should be able to decide which throughput-delay compromise they prefer. In this setting, it appears achievable to match or even exceed CoDel’s performance without modifying gateways.

Figure 2-10: Average utilization and delay of each scheme. Utilization is the average fraction of the cellular link's maximum capacity that the scheme achieved.

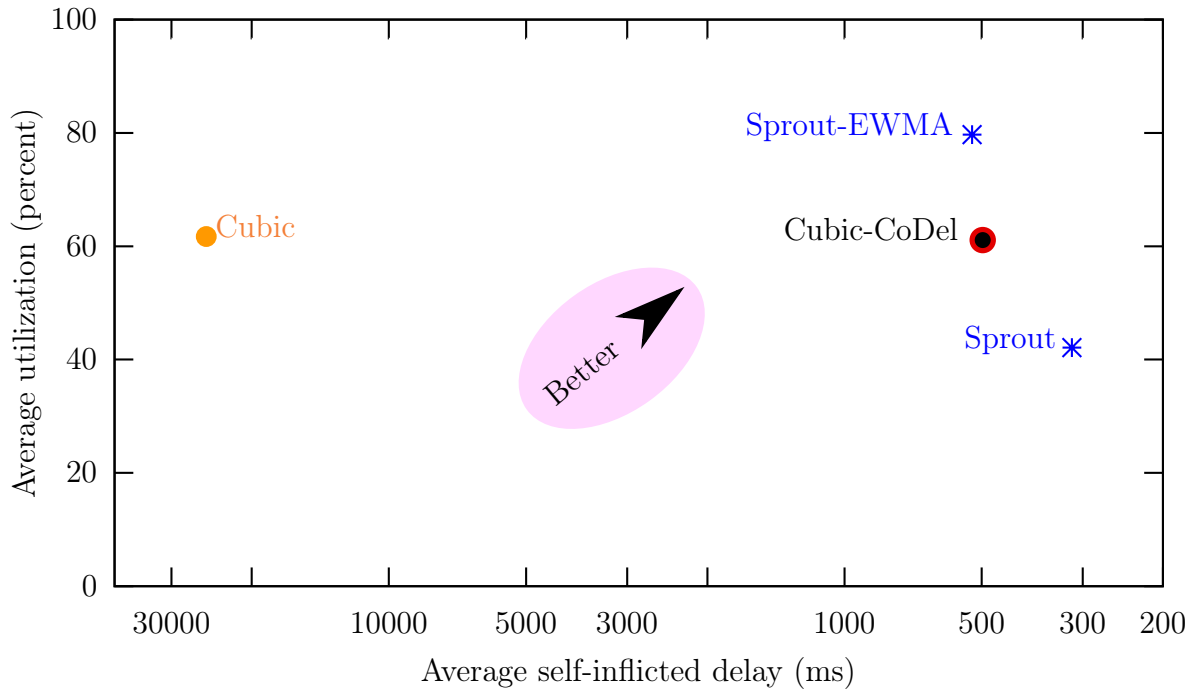
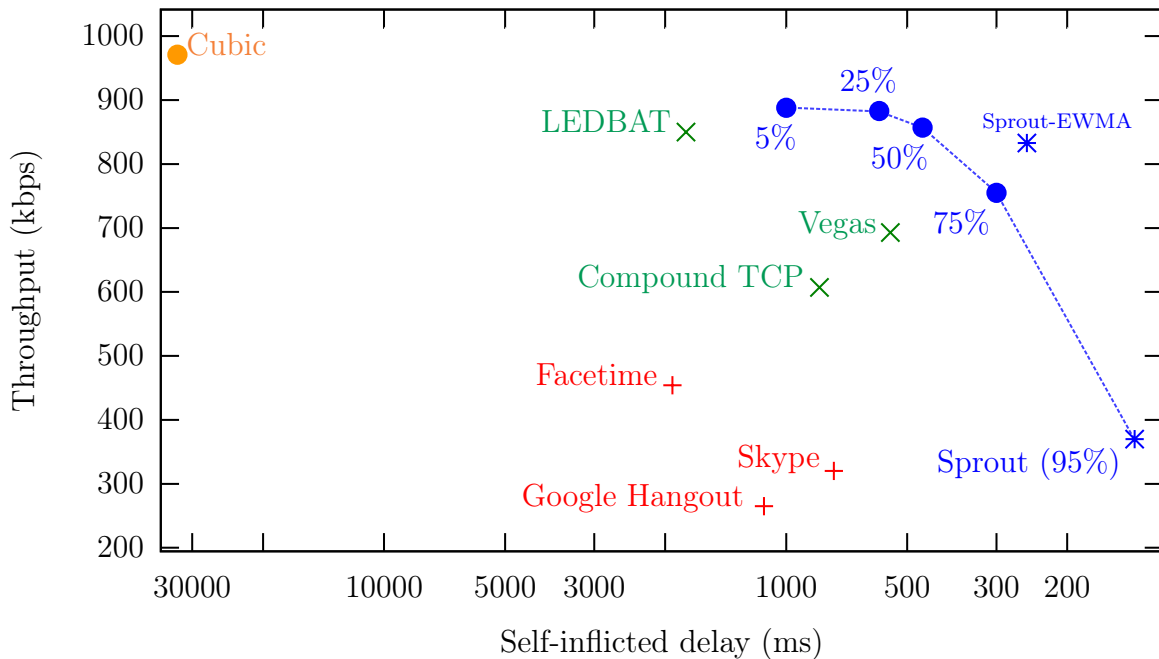


Figure 2-11: Lowering the forecast's confidence parameter allows greater throughput at the cost of more delay. Results on the T-Mobile 3G (UMTS) uplink:



2.4.5 Effect of confidence parameter

The Sprout receiver makes forecasts of a lower bound on how many packets will be delivered with at least 95% probability. We explored the effect of lowering this confidence parameter to express a greater willingness that packets be queued for longer than the sender’s 100 ms tolerance.

Results on one network path are shown in Figure 2-11. The different confidence parameters trace out a curve of achievable throughput-delay tradeoffs. As expected, decreasing the amount of caution in the forecast allows the sender to achieve greater throughput, but at the cost of more delay. Interestingly, although Sprout achieves higher throughput and lower delay than Sprout-EWMA by varying the confidence parameter, it never achieves both at the same time. Why this is—and whether Sprout’s stochastic model can be further improved to beat Sprout-EWMA simultaneously on both metrics—will need to be the subject of further study.

2.4.6 Loss resilience

The cellular networks we experimented with all exhibited low packet loss rates, but that will not be true in general. To investigate the loss resilience of Sprout, we used the traces collected from one network (Verizon LTE) and simulated Bernoulli packet losses (tail drop) with two different packet loss probabilities, 5% and 10% (in each direction). The results are shown in the table below:

Protocol	Throughput (kbps)	Delay (ms)
Downlink		
Sprout	4741	73
Sprout-5%	3971	60
Sprout-10%	2768	58
Uplink		
Sprout	3703	332
Sprout-5%	2598	378
Sprout-10%	1163	314

As expected, the throughput does diminish in the face of packet loss, but Sprout continues to provide good throughput even at high loss rates. (TCP, which interprets loss as a congestion signal, generally suffers unacceptable slowdowns in the face of 10% each-way packet loss.) These results demonstrate that Sprout is relatively resilient to packet loss.

2.4.7 Sprout as a tunnel for competing traffic

We tested whether SproutTunnel, used as a tunnel over the cellular link to a well-connected relay, can successfully isolate bulk-transfer downloads from interactive applications.

We ran two flows: a TCP Cubic bulk transfer (download only) and a two-way Skype videoconference, using the Linux version of Skype.

We compared the situation of these two flows running directly over the emulated Verizon LTE link, versus running them through SproutTunnel over the same link. The experiments lasted about ten minutes each.⁸

	Direct	via Sprout	Change
Cubic throughput	8336 kbps	3776 kbps	−55%
Skype throughput	78 kbps	490 kbps	+528%
Skype 95% delay	6.0 s	0.17 s	−97%

The results suggest that interactive applications can be greatly aided by running their traffic—and any concurrent bulk transfers—through Sprout. Without Sprout to mediate, Cubic squeezes out Skype and builds up huge delays. However, Sprout’s conservatism about delay substantially reduces Cubic’s throughput.

2.5 Sprout in context: the protocol-design contest

In March and April of 2013, we ran a contest in MIT’s graduate course in computer networks to develop the “best” control protocol for a single long flow running over a cellular network. Over a two-week period, we asked students to develop algorithms that maximized the throughput divided by the delay, also known as the “power” of a congestion-control algorithm.

Forty students participated in the contest. At the outset, we gave students a “training” trace from the Verizon LTE network and Sprout’s source code, and set up a dynamic leaderboard that showed the best performance of every student thus far, as well as results from Skype and Sprout.⁹ During the course of the contest, the students made almost 3,000 attempts to maximize power over the Verizon trace.

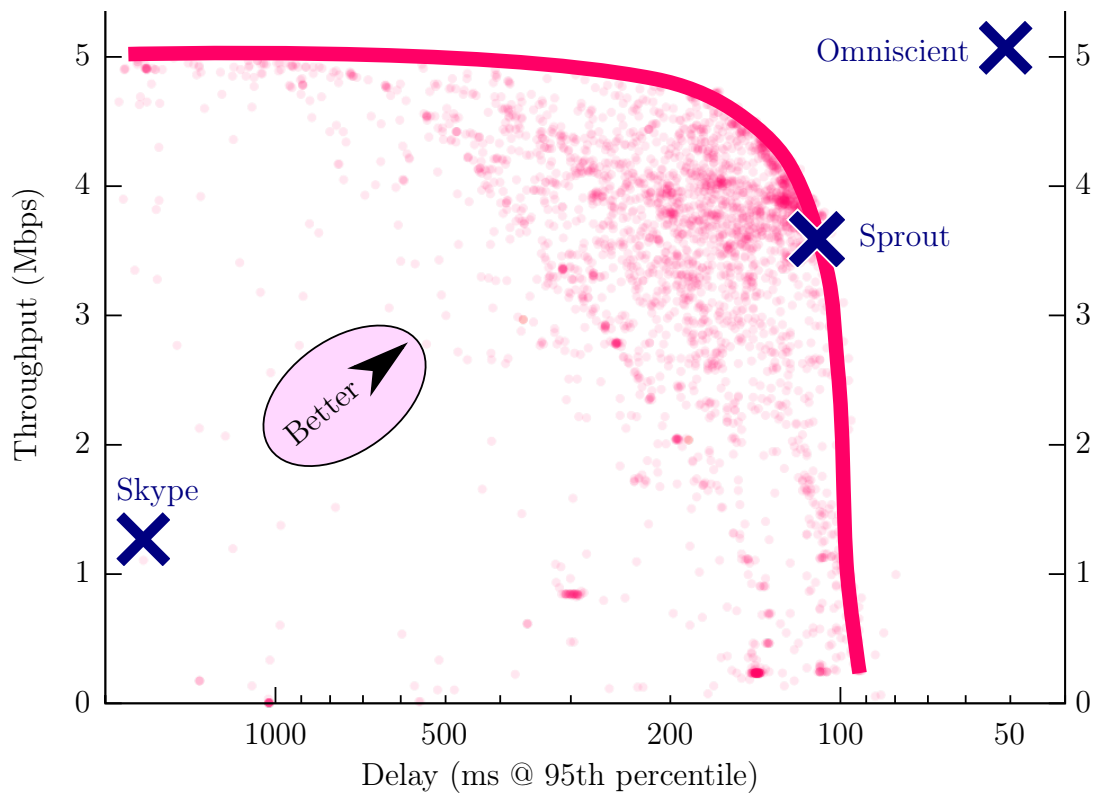
At the end of the two-week period, we froze the students’ final submitted algorithms and collected a fresh trace with the Saturator, again while driving in greater Boston. The final results of the contest were determined by running each of the submissions, and Sprout, over the newly-collected trace. Teams that performed well on this final evaluation were rewarded with gift certificates, and two top-performing teams became co-authors on a forthcoming journal publication describing the contest results and their winning algorithms [53].

Figure 2-12 shows the performance of all of the nearly 3,000 student-generated algorithms on the training trace. There can be no assurance that the algorithms actually map out the range of achievable throughput-delay tradeoffs. Our only rigorous outer bound is the point labeled “Omniscient,” which represents the best possible performance given foreknowledge of the link, so that packets arrive at the queue exactly when the link is able to dequeue and deliver the packet.

⁸In each run, Skype ended the video portion of the call once and was restarted manually.

⁹We also provided students with a second training trace to evaluate the robustness of their candidate algorithms.

Figure 2-12: Sprout, Skype, and 2,994 student-generated protocols running over an emulated Verizon LTE network trace. Sprout is near the efficient frontier determined empirically by the 40 students who participated in the contest. “Omniscient” represents the best achievable performance given foreknowledge of the times when packets will be delivered from the bottleneck queue.



Nonetheless, if we treat the student submissions—trained on this trace—as a proxy for the realizable range of throughput-delay tradeoffs, Sprout was near the efficient frontier, despite having been frozen before the trace was collected.

The results suggest that there may be an unavoidable tradeoff between throughput and delay on this problem, and therefore no “one size fits all” solution. We explore this notion further in Chapter 3.

2.6 Related work

End-to-end algorithms. Traditional congestion-control algorithms generally do not simultaneously achieve high utilization and low delay over paths with high rate variations. Early TCP variants such as Tahoe and Reno [28] do not explicitly adapt to delay (other than from ACK clocking), and require an appropriate buffer size for good performance. TCP Vegas [12], FAST [61], and Compound TCP [56] incorporate round-trip delay explicitly, but the adaptation is reactive and does not directly involve the receiver’s observed rate.

LEDBAT [51] (and TCP Nice [59]) share our goals of high throughput without introducing long delays, but LEDBAT does not perform as well as Sprout. We believe this is because of its choice of congestion signal (one-way delay) and the absence of forecasting. Some recent work proposes TCP receiver modifications to combat bufferbloat in 3G/4G wireless networks [30]. Schemes such as “TCP-friendly” equation-based rate control [20] and binomial congestion control [8] exhibit slower transmission rate variations than TCP, and in principle could introduce lower delay, but perform poorly in the face of sudden rate changes [9].

Google has proposed a congestion-control scheme [39] for the WebRTC system that uses an arrival-time filter at the receiver, along with other congestion signals, to decide whether a real-time flow should increase, decrease, or hold its current bit rate.

Active queue management. Active queue management schemes such as RED [21] and its variants, BLUE [18], AVQ [36], etc., drop or mark packets using local indications of upcoming congestion at a bottleneck queue, with the idea that endpoints react to these signals before queues grow significantly. Over the past several years, it has proven difficult to automatically configure the parameters used in these algorithms. To alleviate this shortcoming, CoDel [44] changes the signal of congestion from queue length to the delay experienced by packets in a queue, with a view toward controlling that delay, especially in networks with deep queues (“bufferbloat” [22]).

Our results show that Sprout largely holds its own with CoDel over challenging wireless conditions without requiring any gateway modifications. It is important to note that network paths in practice have several places where queues may build up (in LTE infrastructure, in baseband modems, in IP-layer queues, near the USB interface in tethering mode, etc.), so one may need to deploy CoDel at all these locations, which could be difficult. However, in networks where there is a lower degree of isolation between queues than the cellular networks we study, CoDel may be the right approach to controlling delay while providing good throughput, but it is a “one-size-fits-all”

method that assumes that a single throughput-delay tradeoff is right for all traffic.

2.7 Limitations

Although Sprout’s results are encouraging, there are several limitations. First, as noted in §2.1 and §2.2, an end-to-end system like Sprout cannot control delays when the bottleneck link also carries aggressive competing traffic that shares the same queue. If a device uses traditional TCP outside of Sprout, the incurred queueing delay—seen by Sprout and every flow—will be substantial.

The accuracy of Sprout’s forecasts depends on whether the application is providing offered load sufficient to saturate the link. For applications that switch intermittently on and off, or don’t desire high throughput, the transient behavior of Sprout’s forecasts (e.g. ramp-up time) becomes more important. Our evaluation assumed that a video coder can meet Sprout’s contract by supplying on demand, with no latency, a newly-coded video frame whose compressed length was exactly equal to the desired amount. In practice, actual video coders will not be able to meet this contract perfectly, and the resulting consequences will need to be evaluated.

We have tested Sprout only in trace-based emulation of eight cellular links recorded in the Boston area in 2012. Although Sprout’s model was frozen before data were collected and was not “tuned” in response to any particular network, we cannot know how generalizable Sprout’s algorithm is without more real-world testing.

Sprout is not a traditional congestion-control protocol, in that it is designed to adapt to varying link conditions, not varying cross traffic. In a cellular link where users have their own queues on the base station, interactive performance will likely be best when the user runs bulk and interactive traffic *inside* Sprout (e.g. using SproutTunnel), not alongside Sprout.

For the classical problem of multiple independent users sharing the same bottleneck, Sprout is insufficient to provide good results. In the rest of this dissertation, we expand on Sprout’s approach to address the “multi-agent” congestion-control problem.

Chapter 3

TCP ex Machina: Computer-Generated Congestion Control

Sprout (Chapter 2) addressed a particular problem in congestion control: the compromise between throughput and delay on a cellular network, where the user has his or her own queue at the bottleneck but does not know the capacity of the link.

The classical problem of distributed resource allocation on the Internet, however, is more challenging than this. Algorithms like TCP NewReno [25] and Cubic [23] are designed for the problem of *multi-agent* congestion control. In general, multiple independent users may contend for the same bottleneck—without knowing for certain how many other users they are contending with.

Is it possible, then, for a computer to design an algorithm from first principles and “discover” the right rules for congestion control? We found that computers can design schemes that in some cases surpass the best human-designed methods to date, when supplied with the appropriate criteria by which to judge a congestion-control algorithm. We attempt to probe the limits of these machine-generated protocols, and discuss how this style of transport-layer protocol design can give more freedom to network architects and link-layer designers.

Congestion control is a fundamental problem in multi-user computer networks. Any proposed solution must answer the question: when should an endpoint transmit each packet of data? An ideal scheme would transmit a packet whenever capacity to carry the packet was available, but because there are many concurrent senders and the network experiences variable delays, this question isn’t an easy one to answer. On the Internet, the past thirty years have seen a number of innovative and influential answers to this question, with solutions embedded at the endpoints (mainly in TCP) aided occasionally by queue management and scheduling algorithms in bottleneck routers that provide signals to the endpoints.

This area has continued to draw research and engineering effort because new link technologies and subnetworks have proliferated and evolved. For example, the past few years have seen an increase in wireless networks with variable bottleneck rates; datacenter networks with high rates, short delays, and correlations in offered load;

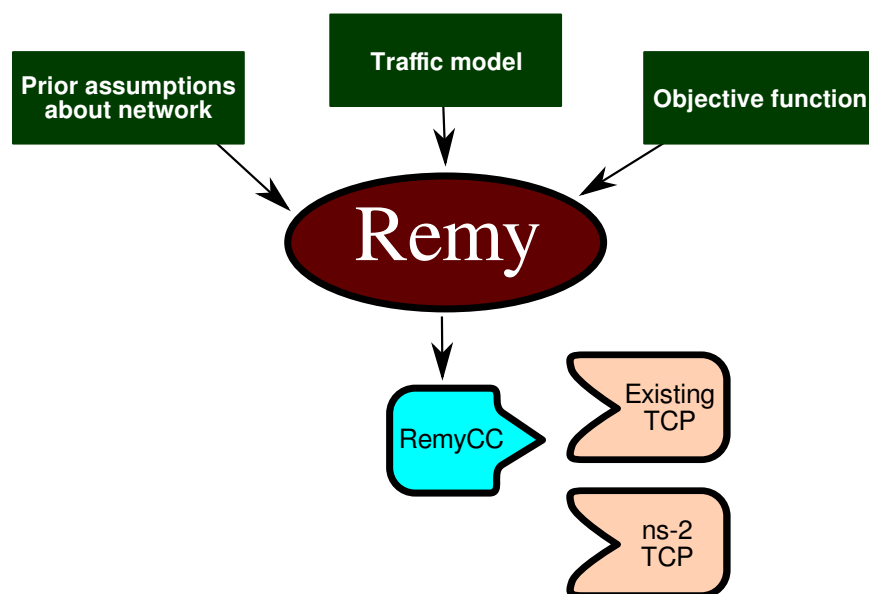


Figure 3-1: Remy designs congestion-control schemes automatically to achieve desired outcomes. The algorithms it produces may replace the congestion-control module of a TCP implementation, and fit into a network library or kernel module that implements congestion control (DCCP, SCTP, the congestion manager, application-layer transmission control libraries, ns-2 modules, etc.).

paths with excessive buffering (now called “bufferbloat”); cellular wireless networks with highly variable, self-inflicted packet delays; links with non-congestive stochastic loss; and networks with large bandwidth-delay products. In these conditions, the classical congestion-control methods embedded in TCP can perform poorly, as many papers have shown (§3.3).

Without the ability to adapt its congestion-control algorithms to new scenarios, TCP’s inflexibility constrains architectural evolution [63]. Subnetworks and link layers are typically evaluated based on how well TCP performs over them. This scorecard can lead to perverse behavior, because TCP’s network model is limited. For example, because TCP assumes that packet losses are due to congestion and reduces its transmission rate in response, some subnetwork designers have worked hard to hide losses. This often simply adds intolerably long packet delays. One may argue that such designs are misguided, but the difficulties presented by “too-reliable” link layers have been a perennial challenge for 25 years [15] and show no signs of abating. With the rise of widespread cellular connectivity, these behaviors are increasingly common and deeply embedded in deployed infrastructure.

The designers of a new subnetwork may well ask what they should do to make TCP perform well. This question is surprisingly hard to answer, because the so-called teleology of TCP is unknown: exactly what objective does TCP congestion control optimize? TCP’s dynamic behavior, when competing flows enter and leave the network, remains challenging to explain [9]. In practice, the need to “make TCP perform well”

is given as a number of loose guidelines, such as IETF RFC 3819 [31], which contains dozens of pages of qualitative best current practice. The challenging and subtle nature of this area means that the potential of new subnetworks and network architectures is often not realized.

3.1 Design overview

We start by explicitly stating an *objective* for congestion control; for example, given an unknown number of users, we may optimize some function of the per-user throughput and packet delay, or a summary statistic such as average flow completion time. Then, instead of writing down rules by hand for the endpoints to follow, we start from the desired objective and work backwards in three steps:

1. First, model the protocol’s prior assumptions about the network; i.e., the “design range” of operation. This model may be different, and have different amounts of uncertainty, for a protocol that will be used exclusively within a data center, compared with one intended to be used over a wireless link or one for the broader Internet. A typical model specifies upper and lower limits on the bottleneck link speeds, non-queueing delays, queue sizes, and degrees of multiplexing.
2. Second, define a traffic model for the offered load given to endpoints. This may characterize typical Web traffic, video conferencing, batch processing, or some mixture of these. It may be synthetic or based on empirical measurements.
3. Third, use the modeled network scenarios and traffic to design a congestion-control algorithm that can later be executed on endpoints.

I developed an optimization tool called Remy that takes these models as input, and designs a congestion-control algorithm that tries to maximize the total expected value of the objective function, measured over the set of network and traffic models. The resulting pre-calculated, optimized algorithm is then run on actual endpoints; no further learning happens after the offline optimization. The optimized algorithm is run as part of an existing TCP sender implementation, or within any congestion-control module. No receiver changes are necessary (as of now).

3.2 Summary of results

Running on an 80-core server at MIT, Remy generally takes between a few hours to a few days of wall-clock time (up to a CPU-year) to generate congestion-control algorithms offline that work on a wide range of network conditions.

Our main results from several simulation experiments with Remy are as follows:

1. For networks broadly consistent with the assumptions provided to Remy at design time, the machine-generated algorithms dramatically outperform existing methods, including TCP Cubic, Compound TCP, and TCP Vegas.

2. Comparing Remy’s algorithms with schemes that require modifications to network gateways, including Cubic-over-sfqCoDel and XCP, Remy generally matched or surpassed these schemes, despite being entirely end-to-end.

On a simulated 15 Mbps fixed-rate link with eight senders contending and an RTT of 150 ms, a computer-generated congestion-control algorithm achieved the following improvements in median throughput and reductions in median queueing delay over these existing protocols:

Protocol	Median speedup	Median delay reduction
Compound	2.1×	2.7×
NewReno	2.6×	2.2×
Cubic	1.7×	3.4×
Vegas	3.1×	1.2×
Cubic/sfqCoDel	1.4×	7.8×
XCP	1.4×	4.3×

In a trace-driven simulation of the Verizon LTE downlink with four senders contending, the *same* computer-generated protocol achieved these speedups and reductions in median queueing delay:

Protocol	Median speedup	Median delay reduction
Compound	1.3×	1.3×
NewReno	1.5×	1.2×
Cubic	1.2×	1.7×
Vegas	2.2×	0.44×
Cubic/sfqCoDel	1.3×	1.3×
XCP	1.7×	0.78×

The source code for Remy, our ns-2 models, the algorithms that Remy designed, and instructions for replicating the experiments and reproducing the figures in this chapter are available from <http://mit.edu/remy>.

3.3 Related work

Starting with Ramakrishnan and Jain’s DECBT scheme [49] and Jacobson’s TCP Tahoe (and Reno) algorithms [28], congestion control over heterogeneous packet-switched networks has been an active area of research. End-to-end algorithms typically compute a congestion window (or, in some cases, a transmission rate) as well as the round-trip time (RTT) using the stream of acknowledgments (ACKs) arriving from the receiver. In response to congestion, inferred from packet loss or, in some cases, rising delays, the sender reduces its window; conversely, when no congestion is perceived, the sender increases its window.

There are many different ways to vary the window. Chiu and Jain [13] showed that among linear methods, additive increase / multiplicative decrease (AIMD) converges

to high utilization and a fair allocation of throughputs, under some simplifying assumptions (long-running connections with synchronized and instantaneous feedback). Our work relaxes these assumptions to handle flows that enter and leave the network, and users who care about latency as well as throughput. Remy’s algorithms are not necessarily linear, and can use both a window and a rate pacer to regulate transmissions.

In evaluating Remy, we compare its output—the computer-generated algorithms—with several end-to-end schemes, including NewReno [25], Vegas [12], Compound TCP [56], Cubic [23], and DCTCP for datacenters [3]. NewReno has the same congestion-control strategy as Reno—slow start at the beginning, on a timeout, or after an idle period of about one retransmission timeout (RTO), additive increase every RTT when there is no congestion, and a one-half reduction in the window on receiving three duplicate ACKs (signaling packet loss). We compare against NewReno rather than Reno because NewReno’s loss recovery is better.

Brakmo and Peterson’s Vegas is a delay-based algorithm, motivated by the insight from Jain’s CARD scheme [29] and Wang and Crowcroft’s DUAL scheme [60] that increasing RTTs may be a congestion signal. Vegas computes a BaseRTT, defined as the RTT in the absence of congestion, and usually estimated as the first RTT on the connection before the windows grow. The expected throughput of the connection is the ratio of the current window size and BaseRTT, if there is no congestion; Vegas compares the *actual* sending rate, and considers the difference, *diff*, between the expected and actual rates. Depending on this difference, Vegas either increases the congestion window linearly ($diff < \alpha$), reduces it linearly ($diff > \beta$), or leaves it unchanged.

Compound TCP [56] combines ideas from Reno and Vegas: when packet losses occur, it uses Reno’s adaptation, while reacting to delay variations using ideas from Vegas. Compound TCP is more complicated than a straightforward hybrid of Reno and Vegas; for example, the delay-based window adjustment uses a binomial algorithm [8]. Compound TCP uses the delay-based window to identify the absence of congestion rather than its onset, which is a key difference from Vegas.

Rhee and Xu’s Cubic algorithm is an improvement over their previous work on BIC [68]. Cubic’s growth is independent of the RTT (like H-TCP [38]), and depends only on the packet loss rate, incrementing as a cubic function of “real” time. Cubic is known to achieve high throughput and fairness independent of RTT, but it also aggressively increases its window size, inflating queues and bloating RTTs (see §3.6).

Other schemes developed in the literature include equation-based congestion control [20], binomial control [8], FastTCP [61], HSTCP, and TCP Westwood [41].

End-to-end control may be improved with explicit router participation, as in Explicit Congestion Notification (ECN) [19], VCP [67], active queue management schemes like RED [21], BLUE [18], CHOKe [47], AVQ [36], and CoDel [44] fair queueing, and explicit methods such as XCP [32] and RCP [55]. AQM schemes aim to prevent persistent queues, and have largely focused on reacting to growing queues by marking packets with ECN or dropping them even before the queue is full. CoDel changes the model from reacting to specific average queue lengths to reacting when the delays measured over some duration are too long, suggesting a persistent queue.

Scheduling algorithms isolate flows or groups of flows from each other, and provide weighted fairness between them. In XCP and RCP, routers place information in packet headers to help the senders determine their window (or rate). One limitation of XCP is that it needs to know the bandwidth of the outgoing link, which is difficult to obtain accurately for a time-varying wireless channel.

In §3.6, we compare Remy’s generated algorithm with XCP and with end-to-end schemes running through a gateway with the CoDel AQM and stochastic fair queueing (sfqCoDel).

TCP congestion control was not designed with an explicit optimization goal in mind, but instead allows overall network behavior to emerge from its rules. Kelly et al. present an interpretation of various TCP congestion-control variants in terms of the implicit goals they attempt to optimize [33]. This line of work has become known as Network Utility Maximization (NUM); more recent work has modeled stochastic NUM problems [69], in which flows enter and leave the network. Remy may be viewed as combining the desire for practical distributed endpoint algorithms with the explicit utility-maximization ethos of stochastic NUM.

We note that TCP stacks have adapted in some respects to the changing Internet; for example, increasing bandwidth-delay products have produced efforts to increase the initial congestion window [17, 14], including recent proposals [5, 57] for this quantity to automatically increase on the timescale of months or years. Remy represents an automated means by which TCP’s entire congestion-control algorithm, not just its initial window, could adapt in response to empirical variations in underlying networks.

3.4 Modeling the congestion-control problem

We treat congestion control as a problem of distributed decision-making under uncertainty. Each endpoint that has pending data must decide for itself at every instant: send a packet, or don’t send a packet.

If all nodes knew in advance the network topology and capacity, and the schedule of each node’s present and future offered load, such decisions could in principle be made perfectly, to achieve a desired allocation of throughput on shared links.

In practice, however, endpoints receive observations that only hint at this information. These include feedback from receivers concerning the timing of packets that arrived and detection of packets that didn’t, and sometimes signals, such as ECN marks, from within the network itself. Nodes then make sending decisions based on this partial information about the network.

The Remy approach hinges on being able to evaluate quantitatively the merit of any particular congestion control algorithm, and search for the best algorithm for a given network model and objective function. I describe here our models of the network and cross traffic, and how we ultimately calculate a figure of merit for an arbitrary congestion control algorithm.

3.4.1 Expressing prior assumptions about the network

From a node’s perspective, we treat the network as having been drawn from a stochastic generative process. We assume the network is Markovian, meaning that it is described by some state (e.g. the packets in each queue) and its future evolution will depend only on the current state.

We have parametrized networks on four axes: the speed of bottleneck links, the number of bottlenecks, the propagation delay of the network paths, and the degree of multiplexing, i.e., the number of senders contending for each bottleneck link. We assume that senders have no control over the paths taken by their packets to the receiver.

Depending on the range of networks over which the protocol is intended to be used, a node may have more or less uncertainty about the network’s key parameters. For example, in a data center, the topology, link speeds, and minimum round-trip times may be known in advance, but the degree of multiplexing could vary over a large range. A virtual private network between “clouds” may have more uncertainty about the link speed. A wireless network path may experience less multiplexing, but a large range of transmission rates and round-trip times.

We have observed only weak evidence for a tradeoff between generality and performance; a protocol designed for a broad range of networks may be beaten by a protocol that has been supplied with more specific and accurate prior knowledge. Quantifying and characterizing these kind of tradeoffs is the subject of Chapter 4.

3.4.2 Traffic model

Remy models the offered load as a stochastic process that switches unicast flows between sender-receivers pairs on or off. In a simple model, each endpoint has traffic independent of the other endpoints. The sender is “off” for some number of seconds, drawn from an exponential distribution. Then it switches on for some number of bytes to be transmitted, drawn from an empirical distribution of flow sizes or a closed-form distribution (e.g. heavy-tailed Pareto). While “on,” we assume that the sender will not stall until it completes its transfer.

In traffic models characteristic of data center usage, the off-to-on switches of contending flows may cluster near one another in time, leading to incast. We also model the case where senders are “on” for some amount of time (as opposed to bytes) and seek maximum throughput, as in the case of videoconferences or similar real-time traffic.

3.4.3 Objective function

Resource-allocation theories of congestion control have traditionally employed the alpha-fairness metric to evaluate allocations of throughput on shared links [54]. A flow that receives steady-state throughput of x is assigned a score of:

$$U_\alpha(x) = \frac{x^{1-\alpha}}{1-\alpha}$$

As $\alpha \rightarrow 1$, in the limit $U_1(x)$ becomes $\log x$.

Because $U_\alpha(x)$ is concave for $\alpha > 0$ and monotonically increasing, an allocation that maximizes the total score will prefer to divide the throughput of a bottleneck link equally between flows. When this is impossible, the parameter α sets the tradeoff between fairness and efficiency. For example, $\alpha = 0$ assigns no value to fairness and simply measures total throughput. $\alpha = 1$ is known as proportional fairness, because it will cut one user’s allocation in half as long as another user’s can be more than doubled. $\alpha = 2$ corresponds to minimum potential delay fairness, where the score goes as the negative inverse of throughput; this metric seeks to minimize the total time of fixed-length file transfers. As $\alpha \rightarrow \infty$, maximizing the total $U_\alpha(x)$ achieves max-min fairness, where all that matters is the minimum resource allocations in bottom-up order [54].

Because the overall score is simply a sum of monotonically increasing functions of throughput, an algorithm that maximizes this total is Pareto-efficient for any value of α ; i.e., the metric will always prefer an allocation that helps one user and leaves all other users the same or better. Tan et al. [37] proved that, subject to the requirement of Pareto-efficiency, alpha-fairness is *the* metric that places the greatest emphasis on fairness for a particular α .

Kelly et al. [33] and further analyses showed that TCP approximately maximizes minimum potential delay fairness asymptotically in steady state, if all losses are congestive and link speeds are fixed.

We extend this model to cover dynamic traffic and network conditions. Given a network trace, we calculate the average throughput x of each flow, defined as the total number of bytes received divided by the time that the sender was “on.” We calculate the average round-trip delay y of the connection.

The flow’s score is then

$$U_\alpha(x) - \delta \cdot U_\beta(y),$$

where α and β express the fairness-vs.-efficiency tradeoffs in throughput and delay, respectively, and δ expresses the relative importance of delay vs. throughput.

We emphasize that the purpose of the objective function is to supply a quantitative goal from a protocol-design perspective. It need not (indeed, does not) precisely represent users’ “true” preferences or utilities. In real usage, different users may have different objectives; a videoconference may not benefit from more throughput, or some packets may be more important than others. In Chapter 4, we discuss the problem of how to accommodate diverse objectives on the same network.

3.5 Generating a congestion-control algorithm

The above model may be viewed as a cooperative game that endpoints play. Given packets to transmit (offered load) at an endpoint, the endpoint must decide when to send packets in order to maximize the global objective function. With a particular congestion-control algorithm running on each endpoint, we can calculate each endpoint’s expected score.

In the traditional game-theoretic framework, an endpoint’s decision to send or abstain can be evaluated after fixing the behavior of all other endpoints. An endpoint makes a “rational” decision to send if doing so would improve its expected score, compared with abstaining.

Unfortunately, when greater individual throughput is the desired objective, on a best-effort packet-switched network like the Internet, it is always advantageous to send a packet. In this setting, if every endpoint acted rationally in its own self-interest, the resulting Nash equilibrium would be congestion collapse.¹ This answer is unsatisfactory from a protocol-design perspective, when endpoints have the freedom to send packets when they choose, but the designer wishes to achieve an efficient and equitable allocation of network capacity.

Instead, we believe the appropriate framework is that of *superrationality* [26]. Instead of fixing the other endpoints’ actions before deciding how to maximize one endpoint’s expected score, what is fixed is the common (but as-yet unknown) algorithm run by all endpoints. As in traditional game theory, the endpoint’s goal remains maximizing its own self-interest, but *with the knowledge* that other endpoints are reasoning the same way and will therefore arrive at the same algorithm.

Remy’s job is to find what that algorithm should be. We refer to a particular Remy-designed congestion-control algorithm as a “RemyCC,” which we then implant into an existing sender as part of TCP, DCCP [35], congestion manager [7], or another module running congestion control. The receiver is unchanged (as of now; this may change in the future), but is expected to send periodic ACK feedback.

Formally, we treat the problem of finding the best RemyCC under uncertain network conditions as a search for the best policy for a decentralized partially-observable Markov decision process, or Dec-POMDP [46]. This model originated from operations research and artificial intelligence, in settings where independent agents work cooperatively to achieve some goal. In the case of end-to-end congestion control, endpoints are connected to a shared network that evolves in Markovian fashion. At every time step, the agents must choose between the actions of “sending” or “abstaining,” using observables from their receiver or from network infrastructure.

3.5.1 Compactly representing the sender’s state

In principle, for any given network, there is an *optimal* congestion-control scheme that maximizes the expected total of the endpoints’ objective functions. Such an algorithm would relate (1) the entire history of observations seen thus far (e.g. the contents and timing of every ACK) and (2) the entire history of packets already sent, to the best action at any given moment between sending a new packet or abstaining. However, the search for such an algorithm is likely intractable; on a general Dec-POMDP it is

¹Other researchers have grappled with this problem; for example, Akella et al. [2] studied a restricted game, in which players are forced to obey the same particular flavor of TCP, but with the freedom to choose their additive-increase and multiplicative-decrease coefficients. Even with this constraint, the authors found that the Nash equilibrium is inefficient, unless the endpoints are restricted to run TCP Reno over a drop-tail buffer, in which case the equilibrium is unfair but not inefficient.

NEXP-complete [11].

Instead, we approximate the solution by greatly abridging the sender’s state. A RemyCC tracks three or four features of the history, which it updates each time it receives a new acknowledgment:

1. An exponentially-weighted moving average (EWMA) of the interarrival time between new acknowledgments received (`ack_ewma`), where each new sample contributes 1/8 of the weight of the moving average.
2. In some RemyCCs, where the goal is operation over a broad range of networks, we consider the same feature but with a longer-term moving average, using a weight of 1/256 for the newest sample (`slow_ack_ewma`).
3. An exponentially-weighted moving average of the time between TCP sender timestamps reflected in those acknowledgments (`send_ewma`), again with a weight of 1/8 to each new sample.
4. The ratio between the most recent RTT and the minimum RTT seen during the current connection (`rtt_ratio`).

Together, we call these variables the *RemyCC memory*. It is worth reflecting on these variables, which are the “congestion signals” used by any RemyCC. The value of these signals can be measured empirically, by “knocking out” a signal and examining the resulting effect on performance. We narrowed the memory to this set after examining and discarding quantities like the most-recent RTT sample, the smoothed RTT estimate, and the difference between the long-term EWMA and short-term EWMA of the observed packet rate or RTT. In our experiments, adding extra state variables didn’t improve the performance of the resulting protocol, and each additional dimension slows down the design procedure considerably. But we cannot claim that Remy’s state variables are necessarily optimal for all situations a protocol might encounter. We expect that any group of estimates that roughly summarizes the recent history could form the basis of a workable congestion-control scheme.

We note that a RemyCC’s memory does not include the two factors that traditional TCP congestion-control schemes use: packet loss and RTT. This omission is intentional: a RemyCC that functions well will see few congestive losses, because its objective function will discourage building up queues (bloating buffers will decrease a flow’s score). Moreover, avoiding packet loss as a congestion signal allows the protocol to robustly handle stochastic (non-congestive) packet losses without adversely reducing performance. We avoid giving the sender access to the RTT (as opposed to the RTT ratio), because we do not want it to learn different behaviors for different RTTs.

At the start of each flow, before any ACKs have been received, the memory starts in a well-known all-zeroes initial state. RemyCCs do not keep state from one “on” period to the next, mimicking TCP’s behavior in beginning with slow start every time a new connection is established (it is possible that caching congestion state is a good idea on some paths, but we don’t consider this here). Although RemyCCs do not depend on loss as a congestion signal, they do inherit the loss-recovery behavior of whatever TCP sender they are added to.

3.5.2 RemyCC: Mapping the memory to an action

A RemyCC is defined by how it maps values of the memory to output *actions*. Operationally, a RemyCC runs as a sequence of lookups triggered by incoming ACKs. (The triggering by ACKs is inspired by TCP’s ACK clocking.) Each time a RemyCC sender receives an ACK, it updates its memory and then looks up the corresponding action. It is Remy’s job to pre-compute this lookup table during the design phase, by finding the mapping that maximizes the expected value of the objective function, with the expectation taken over the network model.

Currently, a Remy action has three components:

1. A multiple $m \geq 0$ to the current congestion window (`cwnd`).
2. An increment b to the congestion window (b could be negative).
3. A lower bound $\tau > 0$ milliseconds on the time between successive sends.

If the number of outstanding packets is greater than `cwnd`, the sender will transmit segments to close the window, but no faster than one segment every r milliseconds.

A RemyCC is defined by a set of piecewise-constant *rules*, each one mapping a rectangular region of the three- or four-dimensional memory space to a three-dimensional action. For each acknowledgment received, the RemyCC will update the memory features $\langle \text{ack_ewma}, \text{send_ewma}, \text{rtt_ratio}, \{\text{slow_ack_ewma}\} \rangle$ and then execute the corresponding action $\langle m, b, \tau \rangle$.

3.5.3 Remy’s automated design procedure

The design phase of Remy is an optimization procedure to efficiently construct this state-to-action mapping, or *rule table*. Remy uses simulation of the senders on various sample networks drawn from the network model, with parameters drawn within the ranges of the supplied prior assumptions. These parameters include the link rates, delays, the number of sources, and the on-off distributions of the sources. Offline, Remy evaluates candidate algorithms on millions of randomly generated network configurations.

A single evaluation step consists of drawing somewhere between 16 and 100 network specimens from the network model, then simulating the RemyCC algorithm at each sender for an extended length of time on each network specimen. At the end of the simulation, the objective function for each sender, given by Equation 3.4.3, is totaled to produce an overall figure of merit for the RemyCC. We explore two cases, $\alpha = \beta = 1$ and $\alpha = 2, \delta = 0$. The first case corresponds to proportional throughput and delay fairness, maximizing

$$U = \log(\text{throughput}) - \delta \cdot \log(\text{delay}),$$

with δ specifying the importance placed on delay vs. throughput. The second case

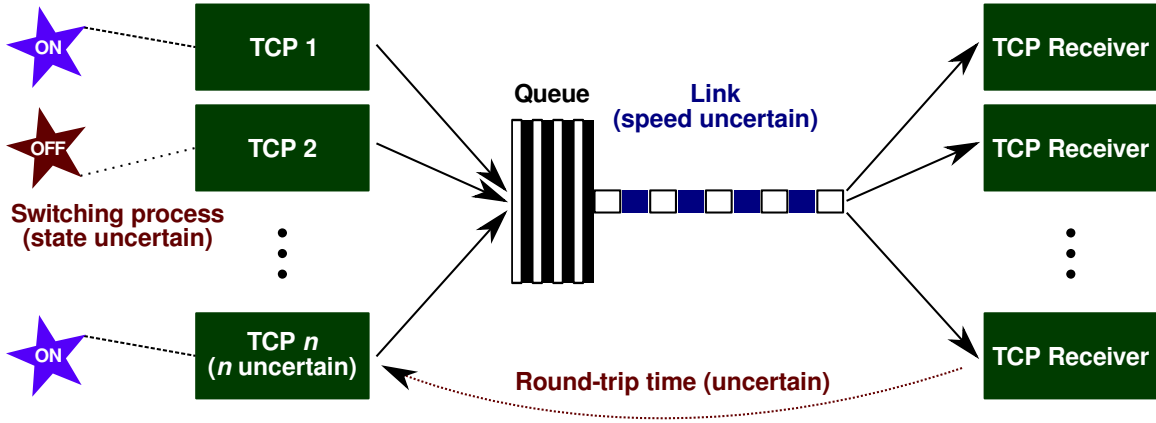


Figure 3-2: Dumbbell network with uncertainty.

corresponds to minimizing the potential delay of a fixed-length transfer, by maximizing

$$U = -\frac{1}{\text{throughput}}.$$

Remy initializes a RemyCC with only a single rule. Any values of the state variables (between 0 and 16,384) are mapped to a default action where $m = 1$, $b = 1$, $r = 0.01$.

Each entry in the rule table has an “epoch.” Remy maintains a global epoch number, initialized to 0. Remy’s search for the “best” RemyCC given a network model is a series of greedy steps to build and improve the rule table:

1. **Set all rules to the current epoch.**
2. **Find the most-used rule in this epoch.** Simulate the current RemyCC and see which rule in the current epoch receives the most use. If no such rules were used, go to step 4.
3. **Improve that action until we can’t anymore.** Focus on this rule and find the best action for it. Draw at least 16 network specimens from the model, and then evaluate roughly 100 candidate increments to the current action, increasing geometrically in granularity as they get further from the current value. For example, evaluate $\tau \pm 0.01$, $\tau \pm 0.08$, $\tau \pm 0.64$, \dots , taking the Cartesian product with the alternatives for m and b .

The modified action is evaluated by substituting it *into all senders* and repeating the simulation in parallel. We use the same random seed and the same set of specimen networks in the simulation of each candidate action to reduce the effects of random variation.

If any of the candidates is an improvement, replace the action with the best new action and repeat the search, still with the same specimen networks and random seed. Otherwise, increment the epoch number of the current rule and go back to step 2.

4. **If we run out of rules in this epoch.** Increment the global epoch. If the new epoch is a multiple of a parameter, K , continue to step 5. Otherwise go back to step 1. We use $K = 4$ to balance structural improvements vs. honing the existing structure.
5. **Subdivide the most-used rule.** Recall that each rule represents a mapping from a rectangular region of memory space to a single action. In this step, find the most-used rule, and the median memory value that triggers it. Split the rule at this point, producing eight new rules (one per dimension of the memory-space), each with the same action as before. Then return to step 1.

By repeating this procedure, the structure of a RemyCC’s rule table becomes an octree [43] or hextree of memory regions. Areas of the memory space more likely to occur receive correspondingly more attention from the optimizer, and are subdivided into smaller bins that yield a more granular function relating memory to action. *Which* rules are more often triggered depends on every endpoint’s behavior as well as the network’s parameters, so the task of finding the right structure for the rule table is best run alongside the process of optimizing existing rules.

3.6 Evaluation

We used ns-2 to evaluate the algorithms generated by Remy and compare them with several other congestion-control methods, including both end-to-end schemes and schemes with router assistance. This section describes the network and workload scenarios and our findings.

3.6.1 Simulation setup and metrics

Congestion-control protocols. The end-to-end schemes we compared with are NewReno, Vegas, Cubic, and Compound. In addition, we compared against two schemes that depend on router assistance: XCP, and Cubic over stochastic fair queueing [42] with each queue running CoDel [44]. We use Nichols’s published sfqCoDel implementation (version released in March 2013) for ns-2.² The Cubic, Compound, and Vegas codes are from the Linux implementations ported to ns-2 and available in ns-2.35. For the datacenter simulation, we also compare with the DCTCP ns-2.35 patch.³

RemyCCs. We used Remy to construct three general-purpose RemyCCs. Each one was designed for an uncertain network model with the dumbbell topology of Figure 3-2, but with three different values of δ (the relative importance of delay): 0.1, 1, and 10. The parameters of the network and traffic model used at design time were:

²<http://www.pollere.net/Txtdocs/sfqcodel.cc>

³<http://www.stanford.edu/~alizade/Site/DCTCP.html>

Quantity	Design range	Distribution
n max senders	1–16	uniform
“on” process	mean 5 s	exponential
“off” process	mean 5 s	exponential
link speed	10–20 Mbps	uniform
round-trip time	100–200 ms	uniform
queue capacity	unlimited	

The model captures a 64-fold range of bandwidth-delay product per user. Each RemyCC took about 3–5 CPU-days to optimize. Calculations were run on Amazon EC2 and on an 80-core and 48-core server at MIT. In wall-clock time, each RemyCC took a few hours to be constructed. The RemyCCs contain between 162 and 204 rules each.

In most experiments, all the sources run the same protocol; in some, we pick different protocols for different sources to investigate how well they co-exist. Each simulation run is generally 100 seconds long, with each scenario run at least 128 times to collect summary statistics.

Workloads. Each source is either “on” or “off” at any point in time. In the evaluation, we modeled the “off” times as exponentially distributed, and the “on” distribution in one of three different ways:

- *by time*, where the source sends as many bytes as the congestion-control protocol allows, for a duration of time picked from an exponential distribution,
- *by bytes*, where the connection sends as many bytes as given by an exponential distribution of a given average and shape, and
- *by empirical distribution*, using the flow-length CDF from a large trace captured in March 2012 and published recently [6]. The flow-length CDF matches a Pareto distribution with the parameters given in Figure 3-3, suggesting that the underlying distribution does not have finite mean. In our evaluation, we add 16 kilobytes to each sampled value to ensure that the network is loaded.

Topologies. We used these topologies in our experiments:

1. **Single bottleneck (“dumbbell”):** The situation in Figure 3-2, with a 1,000-packet buffer, as might be seen in a shared cable-modem uplink. We tested a configuration whose link speed and delay were within the RemyCC design ranges:

Quantity	Range	Distribution
link speed	15 Mbps	exact
round-trip time	150 ms	exact
queue capacity	1000 pkts (tail drop)	

2. **Cellular wireless:** We used the Saturator traces (Section 2.3.1) of the Verizon and AT&T LTE cellular services. We recreated these links within ns-2, queueing

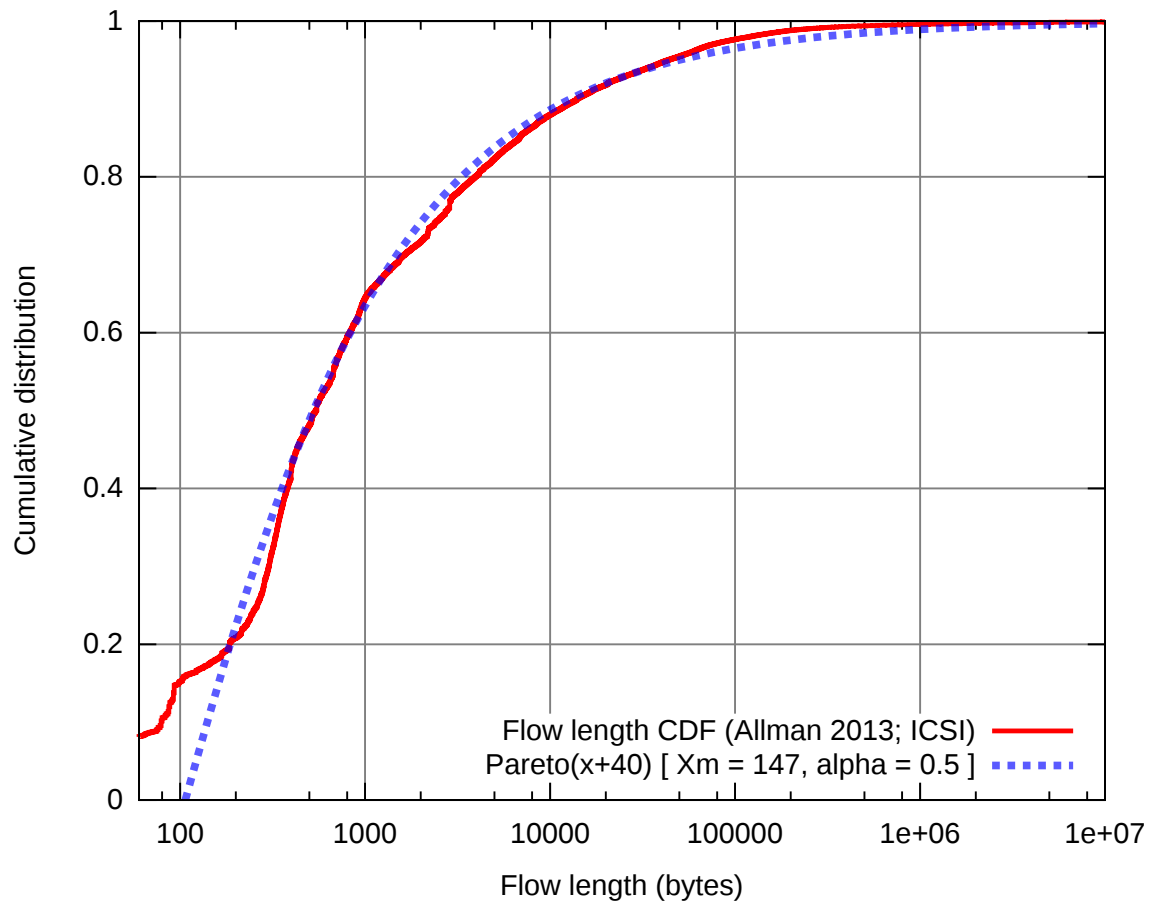


Figure 3-3: Observed Internet flow length distribution matches a Pareto ($\alpha = 0.5$) distribution, suggesting mean is not well-defined.

packets until they are released to the receiver at the same time they were released in the trace. This setup partly probes the RemyCC’s resilience to “model mismatch”—in both the Verizon and AT&T traces, throughput and round-trip time strayed outside the limits of the RemyCC design range.

Quantity	Range	Distribution
link speed	varied 0–50 Mbps	empirical
round-trip time	50 ms	exact
queue capacity	1000 pkts (tail drop)	

3. **Differing RTTs:** Cases where different RemyCCs, contending for the same link, had different RTTs to their corresponding receiver. We analyzed these cases for throughput and delay fairness and compared with existing congestion-control schemes.

Quantity	Range	Distribution
n max senders	4	
“on” process	16×10^3 – 3.3×10^9 bytes	Fig. 3-3
“off” process	mean 0.2 sec	exponential
link speed	10 Mbps	exact
queue capacity	1000 pkts (tail drop)	

4. **Datacenter:** We compared a RemyCC against DCTCP in a simulated data-center topology.

Quantity	Range	Distribution
n max senders	64	exact
“on” process	mean 20 megabytes	exponential
“off” process	mean 0.1 sec	exponential
link speed	10 Gbps	exact
round-trip time	4 ms	exact
queue capacity	1000 pkts (tail drop)	(for RemyCC)
queue capacity	modified RED	(for DCTCP)

Metrics. We measure the throughput and average queueing delay observed for each source-destination pair. With an on-off source, measuring throughput takes some care. We define the throughput of a pair as follows. Suppose the pair is active during (non-overlapping) time intervals of length t_1, t_2, \dots during the entire simulation run of T seconds. If in each interval the protocol successfully receives s_i bytes, we define the throughput for this connection as $\sum s_i / \sum t_i$.

We are interested in the end-to-end delay as well; the reasoning behind Remy’s objective function and the δ parameter is that protocols that fill up buffers to maximize throughput are not as desirable as ones that achieve high throughput *and* low delay—both for their effect on the user, who may prefer to get his packets to the receiver sooner, as well as any other users who share the same FIFO queue.

We present the results for the different protocols as **throughput-delay plots**, where the log-scale x -axis is the queueing delay (average per-packet delay in excess of minimum RTT). Lower, better, delays are to the right. The y -axis is the throughput.

Protocols on the top right are the best on such plots. We take each individual 100-second run from a simulation as one point, and then compute the $1\text{-}\sigma$ elliptic contour of the maximum-likelihood 2D Gaussian distribution that explains the points. To summarize the whole scheme, we plot the median per-sender throughput and queueing delay as a circle.

Ellipses that are narrower in the throughput or delay axis correspond to protocols that are more consistent in allocating those quantities. Protocols with large ellipses are those where identically-positioned users differ widely in experience based on the luck of the draw or the timing of their entry to the network.⁴ The orientation of an ellipse represents the *covariance between the throughput and delay* measured for the protocol; if the throughput were uncorrelated with the queueing delay (note that we show the queueing delay, not the RTT), the ellipse’s axes would be parallel to the graph’s. Because of the variability and correlations between these quantities in practice, we believe that such throughput-delay plots are an instructive way to evaluate congestion-control protocols; they provide more information than simply reporting mean throughput and delay values.

3.6.2 Single-bottleneck results

We start by investigating performance over the simple, classic single-bottleneck “dumbbell” topology. Although it does not model the richness of real-world network paths, the dumbbell is a valuable topology to investigate because in practice there are many single-bottleneck paths experienced by Internet flows.

Recall that this particular dumbbell link had most of its parameters found inside the limits of the design range of the RemyCCs tested. As desired, this test demonstrates that Remy was successful in producing a family of congestion-control algorithms for this type of network.

Results from the 8-sender and 12-sender cases are shown in Figures 3-4 and 3-5. RemyCCs are shown in light blue; the results demonstrate the effect of the δ parameter in weighting the cost of delay. When $\delta = 0.1$, RemyCC senders achieve greater median throughput than those of any other scheme, and the lowest delay (other than the two other RemyCCs). As δ increases, the RemyCCs trace out an achievability frontier of the compromise between throughput and delay. In this experiment, the computer-generated algorithms outperformed all the human-designed ones.

From right to left and bottom to top, the end-to-end TCP congestion-control schemes trace out a path from most delay-conscious (Vegas) to most throughput-conscious (Cubic), with NewReno and Compound falling in between.

The schemes that require in-network assistance (XCP and Cubic-over-sfqCoDel, shown in green) achieve higher throughput than the TCPs, but less than the two

⁴The ellipse represents the variability of the results among all nodes across all simulation runs. Thus, the notion of “variability” here includes both within-run variability or fairness (differing results among nodes that compete concurrently) as well as variability across runs. For the most part, our 100-second simulation runs were long enough, relative to flow durations, that the ellipses can broadly be interpreted as an indicator of fairness.

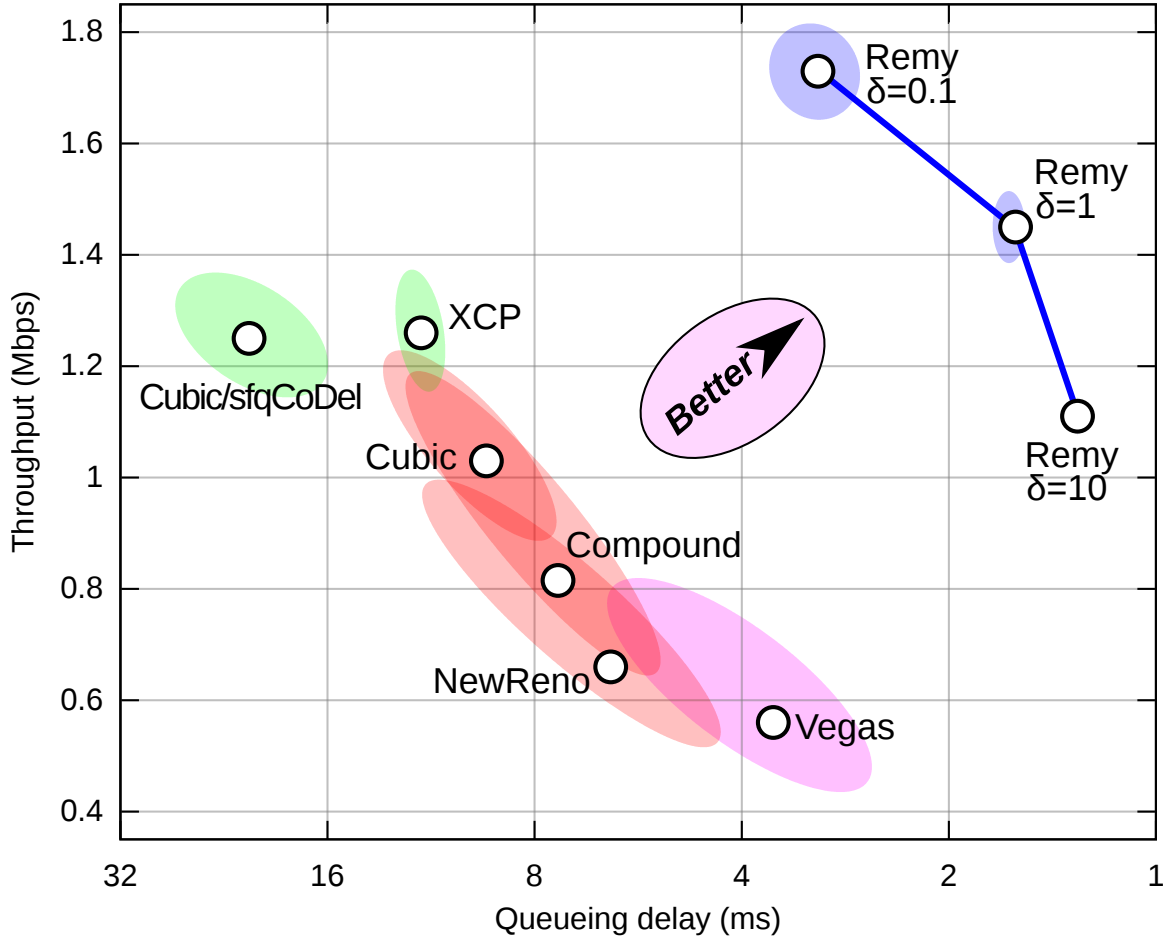


Figure 3-4: Results for each of the schemes over a 15 Mbps dumbbell topology with $n = 8$ senders, each alternating between flows of exponentially-distributed byte length (mean 100 kilobytes) and exponentially-distributed off time (mean 0.5 s). Medians and 1- σ ellipses are shown. The blue line represents the efficient frontier, which here is defined entirely by the RemyCCs.

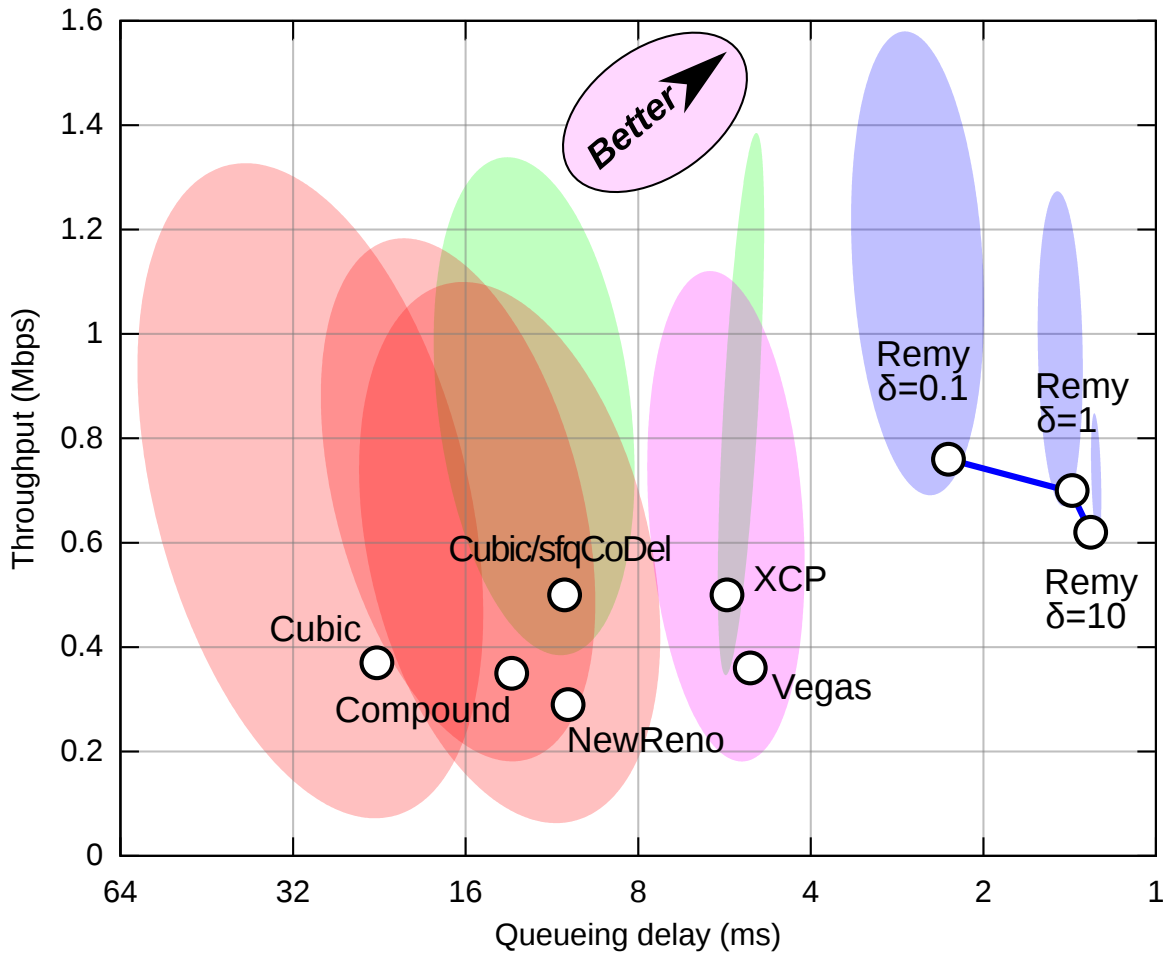


Figure 3-5: Results for the dumbbell topology with $n = 12$ senders, each alternating between flows whose length is drawn from the ICSI trace (Fig. 3-3) and exponentially-distributed off time (mean = 0.2 s). Because of the high variance of the sending distribution, $\frac{1}{2}$ - σ ellipses are down. The RemyCCs again mark the efficient frontier.

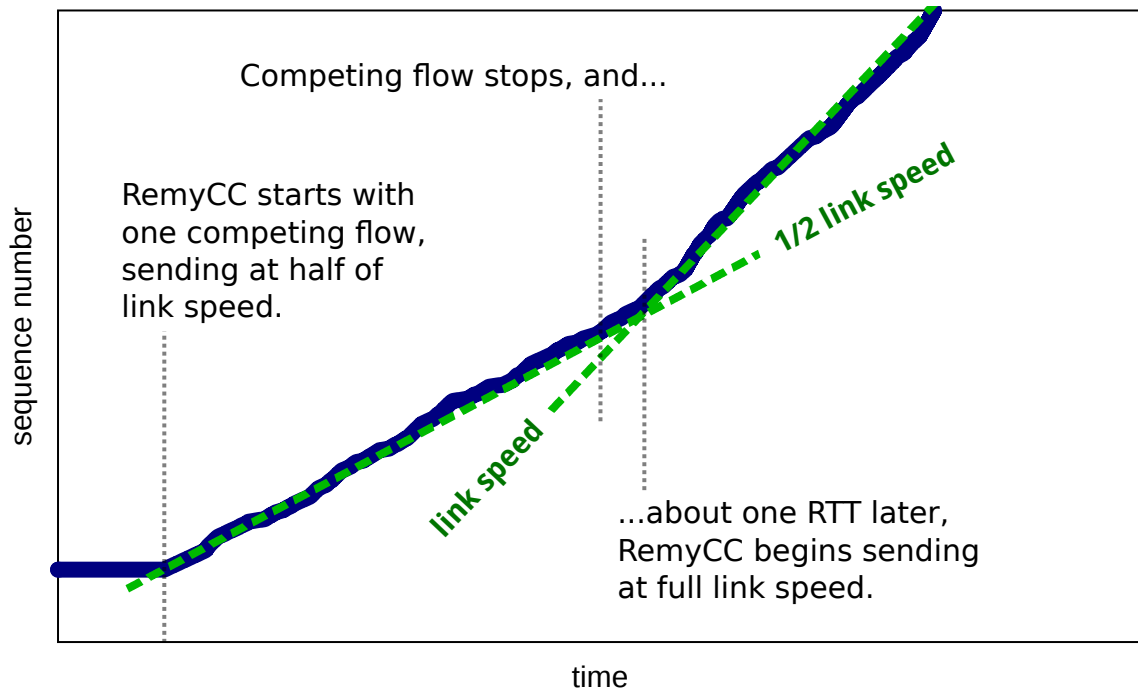


Figure 3-6: Sequence plot of a RemyCC flow in contention with varying cross traffic. The flow responds quickly to the departure of a competing flow by doubling its sending rate.

more throughput-conscious RemyCCs.⁵ This result is encouraging, because it suggests that even a purely end-to-end scheme can outperform well-designed algorithms that involve active router participation. This demonstrates that distributed congestion-control algorithms that explicitly maximize well-chosen objective functions can achieve gains over existing schemes. As we will see in Section 4.4.1, this substantially better performance will not hold when the design assumptions of a RemyCC are contradicted at runtime.

In Figures 3-4 and 3-5, the RemyCCs do not simply have better median performance: they also give more consistent throughput and delay to individual flows. To explain this result, we investigated how multiple RemyCC flows share the network. We found that when a new flow starts, the system converges to an equitable allocation quickly, generally after little more than one RTT. Figure 3-6 shows the sequence of transmissions of a new RemyCC flow that begins while sharing the link. Midway through the flow, the competing traffic departs, allowing the flow to start consuming

⁵It may seem surprising that sfqCoDel, compared with DropTail, *increased* the median RTT of TCP Cubic. CoDel drops a packet at the front of the queue if all packets in the past 100 ms experienced a queueing delay (sojourn time) of at least 5 ms. For this experiment, the transfer lengths are only 100 kilobytes; with a 500 ms “off” time, such a persistent queue is less common even though the mean queueing delay is a lot more than 5 ms. DropTail experiences more losses, so has lower delays (the maximum queue size is $\approx 4\times$ the bandwidth-delay product), but also lower throughput than CoDel. In other experiments with longer transfers, Cubic did experience lower delays when run over sfqCoDel instead of DropTail.

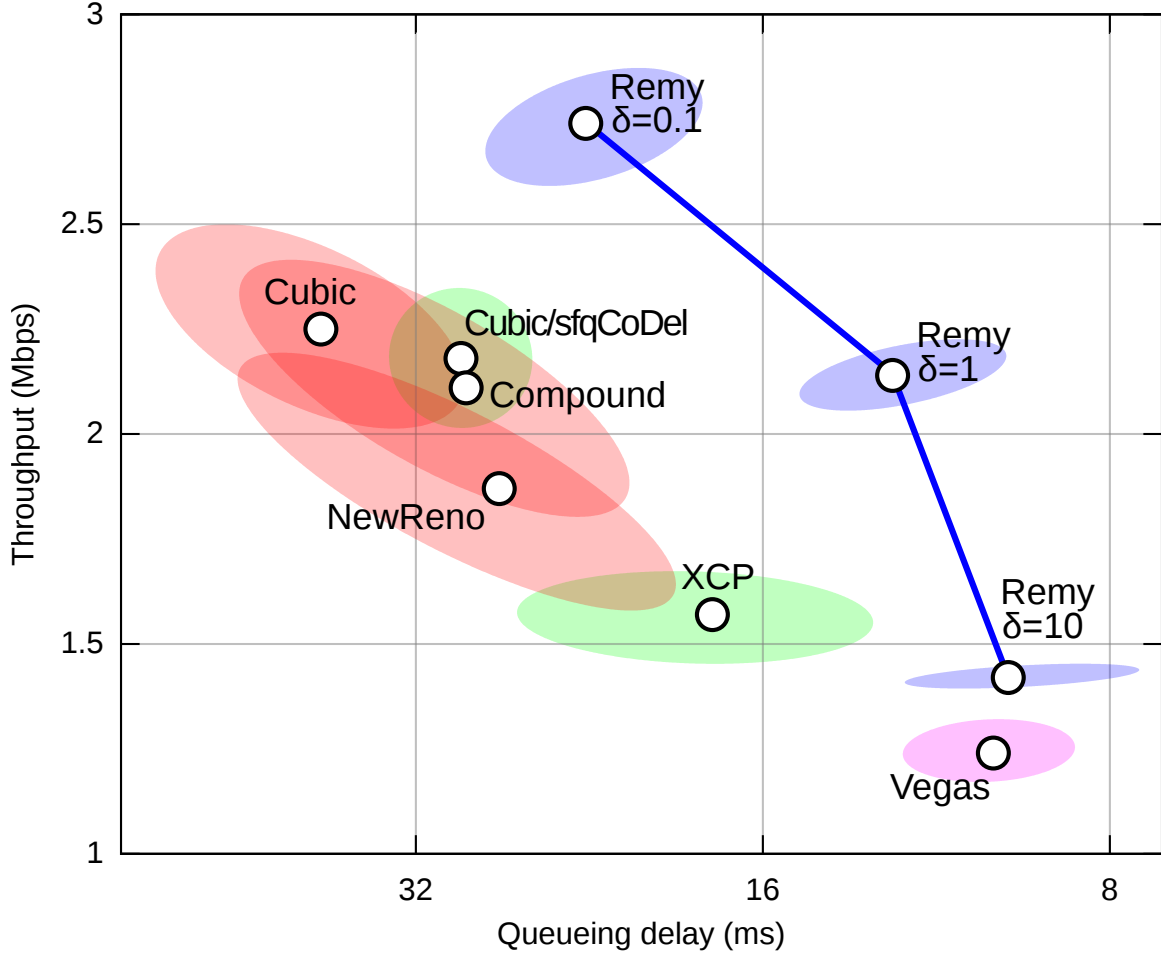


Figure 3-7: Verizon LTE downlink trace, $n = 4$. $1\text{-}\sigma$ ellipses are shown. The RemyCCs define the efficient frontier. Senders alternated between exponentially-distributed file transfers (mean 100 kilobytes) and exponentially-distributed pause times (mean 0.5 s).

the whole bottleneck rate.

3.6.3 Cellular wireless links

Cellular wireless links are tricky for congestion-control algorithms because their link rates vary with time.⁶

By running a program that attempts to keep a cellular link backlogged but without causing buffer overflows, we measured the variation in download speed on Verizon's and AT&T's LTE service while mobile. We then ran simulations over these pre-recorded traces, with the assumption that packets are enqueued by the network until they can be dequeued and delivered at the same instants seen in the trace.

As discussed above, we did not design the RemyCCs to accommodate such a wide variety of throughputs. Running the algorithm over this link illustrated some of the

⁶XCP, in particular, depends on knowing the speed of the link exactly; in our tests on cellular traces we supplied XCP with the long-term average link speed for this value.

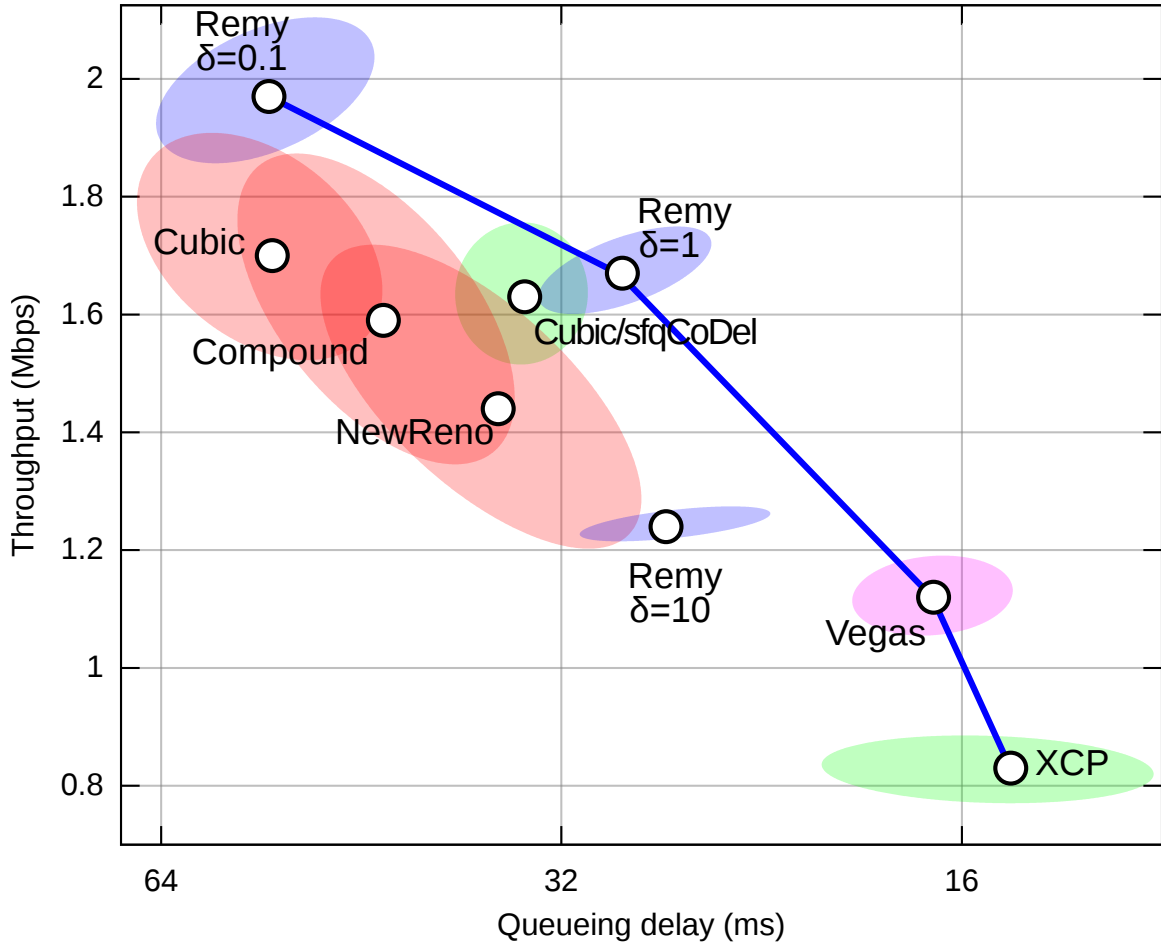


Figure 3-8: Verizon LTE downlink trace, $n = 8$. $1\text{-}\sigma$ ellipses are shown. As the degree of multiplexing increases, the schemes move closer together in performance and router-assisted schemes begin to perform better. Two of the three RemyCCs are on the efficient frontier.

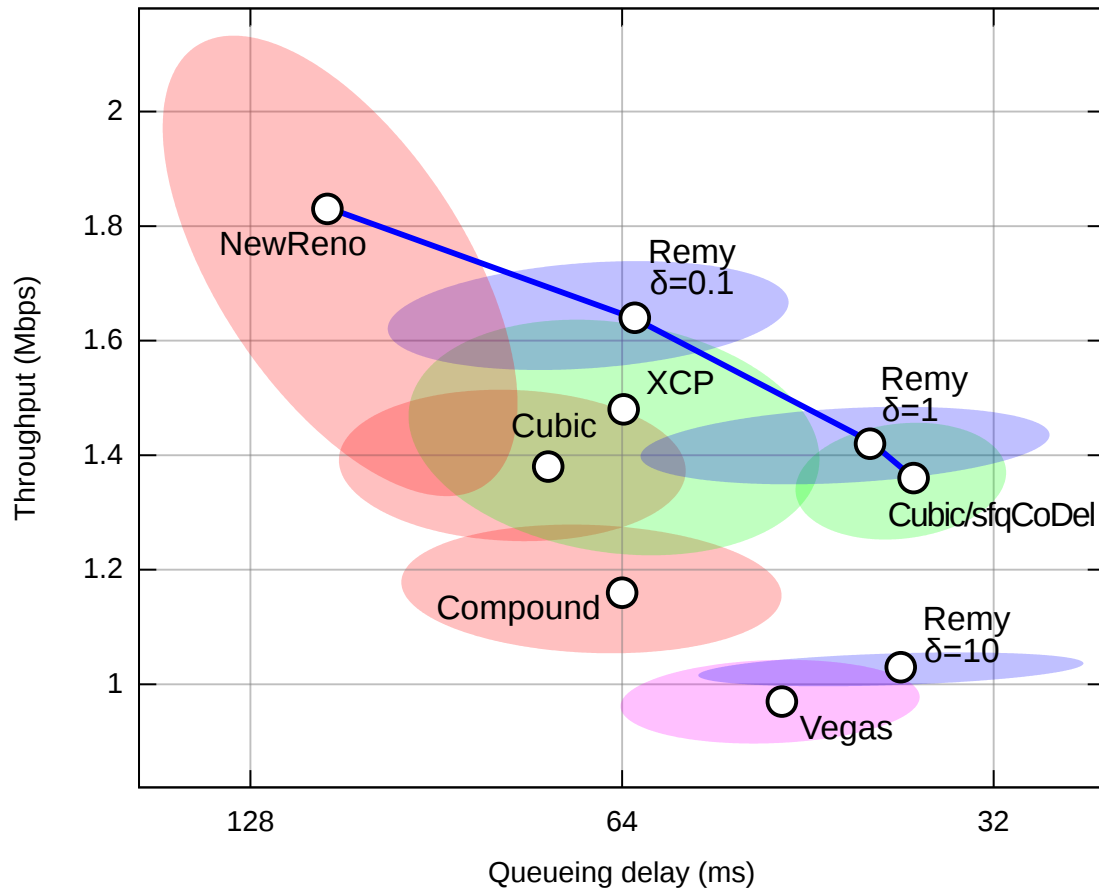


Figure 3-9: AT&T LTE downlink trace, $n = 4$. Two of the RemyCCs are on the efficient frontier.

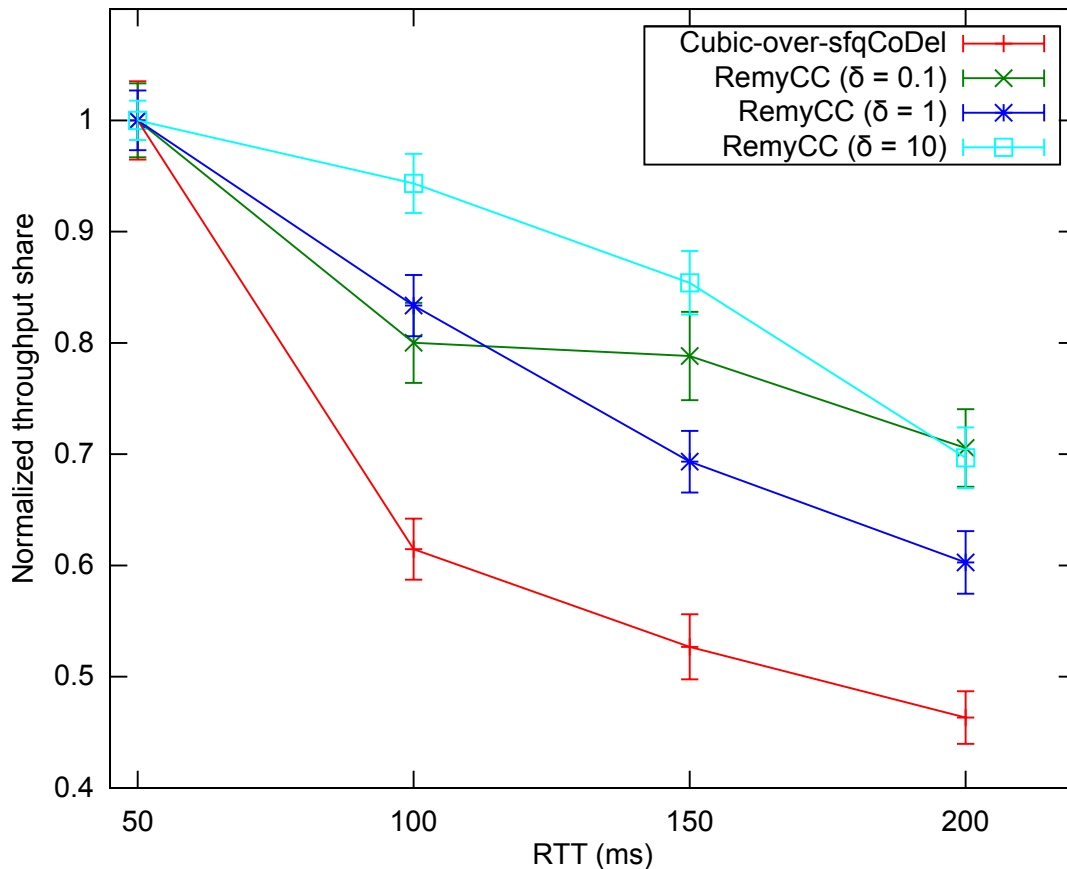


Figure 3-10: The RemyCCs’ RTT unfairness compares favorably to Cubic-over-sfqCoDel. Error bars represent standard error of the mean over 128 100-second simulations.

limits of a RemyCC’s generalizability beyond situations encountered during the design phase.

Somewhat to our surprise, for moderate numbers of concurrent flows, $n \leq 8$, the RemyCCs continued to surpass (albeit narrowly) the best human-designed algorithms, even ones benefiting from in-network assistance. See Figures 3-7 and 3-8.

3.6.4 Differing RTTs

We investigated how the RemyCCs allocate throughput on a contested bottleneck link when the competing flows have different RTTs. At the design stage, all contending flows had the same RTT (which was drawn randomly for each network specimen from between 100 ms and 200 ms), so the RemyCCs were not designed to exhibit RTT fairness explicitly.

We compared the RemyCCs with Cubic-over-sfqCoDel by running 128 realizations of a four-sender simulation where one sender-receiver pair had RTT of 50 ms, one had 100 ms, one 150 ms, and one 200 ms. The RemyCCs did exhibit RTT unfairness, but

more modestly than Cubic-over-sfqCoDel (Fig. 3-10).

3.6.5 Datacenter-like topology

We simulated 64 connections sharing a 10 Gbps datacenter link, and compared DCTCP [3] (using AQM inside the network) against a RemyCC with a 1000-packet tail-drop queue. The RTT of the path in the absence of queueing was 4 ms. Each sender sent 20 megabytes on average (exponentially distributed) with an “off” time between its connections exponentially distributed with mean 100 milliseconds.

We used Remy to design a congestion-control algorithm to maximize $-\frac{1}{\text{throughput}}$ (minimum potential delay) over these network parameters, with the degree of multiplexing assumed to have been drawn uniformly between 1 and 64.

The results for the mean and median throughput (tput) for the 20 megabyte transfers are shown in the following table:

	tput: mean, med	rtt: mean, med
DCTCP (ECN)	179, 144 Mbps	7.5, 6.4 ms
RemyCC (DropTail)	175, 158 Mbps	34, 39 ms

These results show that a RemyCC trained for the datacenter-network parameter range achieves comparable throughput at lower variance than DCTCP, a published and deployed protocol for similar scenarios. The per-packet latencies (and loss rates, not shown) are higher, because in this experiment RemyCC operates over a DropTail bottleneck router, whereas DCTCP runs over an ECN-enabled RED gateway that marks packets when the instantaneous queue exceeds a certain threshold. Developing RemyCC schemes for networks with ECN and AQM is an area for future work.

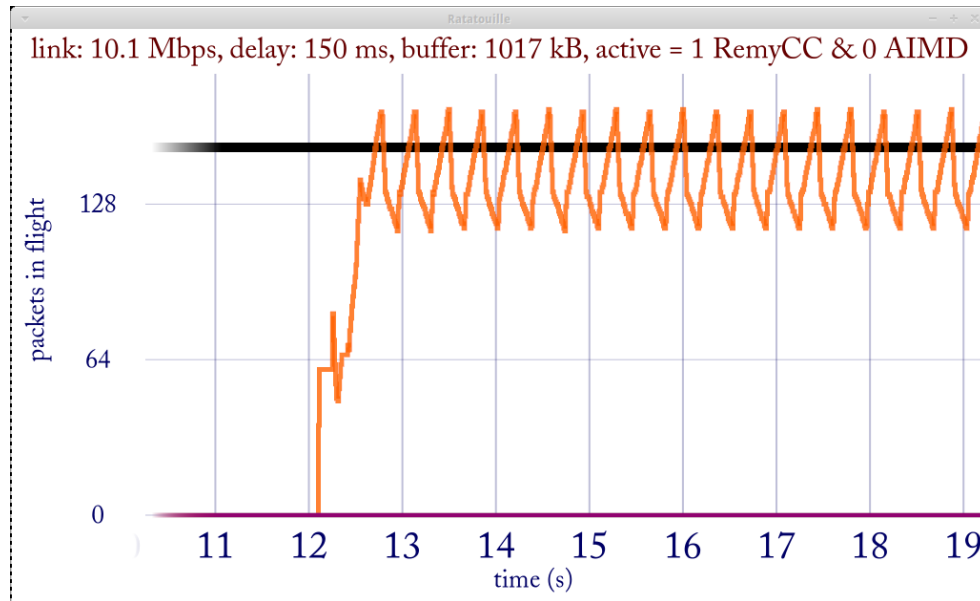
3.7 Ratatouille, a tool for understanding RemyCCs

In our experience, high-performing computer-generated congestion-control algorithms present a different kind of complexity than existing mechanisms. Traditional TCP congestion control schemes specify relatively simple rules for each endpoint, but the resulting emergent behavior of a multiuser network is not easily specified and can be suboptimal and unstable. By contrast, Remy’s algorithms specify hundreds of rules at the endpoints, but produce more consistent overall behavior.

We think this tradeoff is ultimately worth taking: today’s endpoints can execute complex algorithms almost as easily as simple ones, and the *computational* complexity of a RemyCC is low in any event. But when endpoint algorithms become so complex, it is challenging to explain why a particular RemyCC works without considerable effort spent reverse-engineering it. To help with this task, I built a visualization and debugging tool for RemyCCs, called Ratatouille.

Ratatouille displays real-time output from the Remy network simulator, and plots the current number of packets-in-flight for each concurrent flow as it evolves over time. The user adjusts a physical USB control surface (a software mixing board) to vary network parameters, including the number of flows with offered load and the link

Figure 3-11: Ratatouille allows a user to vary network conditions on a physical control surface and observe the resulting behavior of a RemyCC in real-time. Below, the initial startup and steady-state behavior of the RemyCC-100x algorithm of Section 4.4.1.



rate, propagation delay, and buffer size at the bottleneck queue. The user can then observe how the flows respond to the changing conditions. Ratatouille can also run conventional AIMD TCP-like flows for comparison.

By exploring the evolving values of the congestion signals, we have gained some intuition into how a RemyCC can achieve good performance. When the network is in steady state, the congestion signals of a RemyCC sender tend to fall into an orbit, or limit cycle, in which the average number of packets-in-flight over the cycle is close to the “ideal” number: the bandwidth-delay product of the bottleneck link divided by the degree of multiplexing.

The period of the limit cycle is generally a small number of RTTs. By contrast, TCP’s congestion window generally oscillates on a much longer timescale that depends on the buffer size, because the “multiplicative decrease” feature of TCP’s control rules is only triggered when packet loss is detected, typically only after buffer overflow occurs in the bottleneck queue.

Our observations suggest that a RemyCC’s ability to achieve better throughput than TCP on finite-length flows is mostly due to its quicker response to changes in conditions—a RemyCC will generally “ramp up” to the ideal number of packets-in-flight more quickly than the TCP slow-start algorithm (Figure 3-11).

Although Ratatouille has allowed us to make observations that suggest “why” RemyCCs achieve their good performance compared with human-designed mechanisms, much more work will be necessary before we can reason about the performance and robustness of RemyCCs in a rigorous way.

Chapter 4

Using Remy to Measure the Learnability of Congestion Control

The previous chapter described the development of Remy, our automated protocol-design tool. Remy generates congestion-control schemes from first principles, given an uncertain model of the underlying network and a stated application objective to pursue.

In practice, the designer of a congestion-control scheme for the wide-area Internet cannot expect to supply a protocol-design tool with a perfectly faithful model of the actual Internet. An actual model will necessarily be simplified and imperfect. Nonetheless, designers would naturally like to develop protocols that are broadly useful on real networks.

In this chapter, we use Remy to quantify how much tolerance the protocol designer can expect, or in other words: *How easy is it to “learn” a network protocol to achieve desired goals, given a necessarily imperfect model of the networks where it will ultimately be deployed?*

Under this umbrella, we examine a series of questions about what knowledge about the network is important when designing a congestion-control protocol and what simplifications are acceptable:

1. **Knowledge of network parameters.** Is there a tradeoff between the performance of a protocol and the breadth of its intended operating range of network parameters [66]? Will a “one size fits all” protocol designed for networks spanning a thousand-fold range of link rates necessarily perform worse than one targeted at a subset of networks whose parameters can be more precisely defined in advance? (§4.4.1)
2. **Structural knowledge.** How faithfully do protocol designers need to understand the topology of the network? What are the consequences of designing protocols for a simplified network path with a single bottleneck, then running them on networks with more-complicated structure? (§4.4.2)
3. **Knowledge about other endpoints.** In many settings, a newly-designed protocol will need to coexist with traffic from incumbent protocols. What are

the consequences of designing a protocol with the knowledge that some endpoints may be running incumbent protocols whose traffic needs to be treated fairly (e.g. the new protocol needs to divide a contended link evenly with TCP), versus a clean-slate design? (§4.4.3)

Each of the above areas of inquiry is about the effect of a protocol designer’s imperfect understanding of the future network that a decentralized congestion-control protocol will ultimately be run over. In principle, we could quantify such an effect by evaluating, on the actual network, the performance of the “best possible” congestion-control protocol designed for the imperfect network *model*, and comparing that with the best-possible protocol for the actual network itself.

In practice, however, there is no tractable way to calculate the optimal congestion-control protocol for a given network. Instead, throughout this chapter we use Remy to construct achievability results—RemyCCs—that stand in for the “best possible” average performance across the range of network parameters supplied by the designer.

We emphasize that there can be no assurance that Remy’s algorithms actually comes close to the optimal congestion-control protocols, except to the extent that they approach analytical upper bounds on performance and outperform existing schemes across the same networks.

4.1 Summary of results

The principal findings of these experiments are as follows:

Modeling a two-bottleneck network as a single bottleneck hurt performance only mildly. On the two-hop network of Figure 4-2, on average we find that a protocol designed for a simplified, single-bottleneck model of the network underperformed by **17%** a protocol that was designed with full knowledge of the network’s two-bottleneck structure (Figure 4-3). The simplified protocol outperformed TCP Cubic-over-sfqCoDel by $2.75\times$ on average. In this example, full knowledge of the network topology during the design process was not crucial.

We found weak evidence of a tradeoff between operating range and performance. We created a RemyCC designed for a range of networks whose link rates spanned a thousand-fold range between 1 Mbps and 1000 Mbps, as well as three other protocols that were more narrowly-targeted at hundred-fold, ten-fold, and two-fold ranges of link rate (Figure 4-1).

The “thousand-fold” RemyCC achieved close to the peak performance of the “two-fold” RemyCC. Between link rates of 22–44 Mbps, the “thousand-fold” RemyCC achieved **within 3% of the throughput** of the “two-fold” protocol that was designed for this exact range. However, the queueing delay of the “thousand-fold” protocol was **46% higher**, suggesting some benefit from more focused operating conditions. It also takes Remy longer to optimize (offline) a “thousand-fold” RemyCC compared with a two-fold RemyCC. In run-time operation, the computational cost of the two algorithms is similar.

Over the full range of 1 Mbps to 1000 Mbps, the “thousand-fold” RemyCC protocol matched or outperformed TCP Cubic and Cubic-over-sfqCoDel at every link rate (Figure 4-1). This result suggests that it may be possible to construct “one size fits all” end-to-end congestion-control protocols that outperform TCP, even when TCP is assisted by in-network algorithms.

TCP-awareness hurt performance when TCP cross-traffic was absent, but helped dramatically when present. We measured the costs and benefits of “TCP-awareness”—designing a protocol with the explicit knowledge that it may be competing against other endpoints running TCP, compared with a “TCP-naive” protocol for a network where the other endpoints only run the same TCP-naive protocol.

When contending only with other endpoints of the same kind, the “TCP-naive” protocol achieved **55% less queueing delay** than the TCP-aware protocol. In other words, “TCP-awareness” has a cost measured in lost performance when TCP cross-traffic is absent (Figure 4-4).

But when contending against TCP, the “TCP-naive” protocol was squeezed out by the more-aggressive cross-traffic. By contrast, the “TCP-aware” protocol achieved **36% more throughput** and **37% less queueing delay** than the “TCP-naive” protocol, and reached an equitable division of the available capacity with TCP (Figure 4-5).

4.2 Protocol design as a problem of learnability

TCP congestion control was not designed with an explicit objective function in mind. Kelly et al. present an interpretation of TCP congestion-control variants in terms of the implicit goals they attempt to optimize [33]. This line of work is known as Network Utility Maximization (NUM); more recent work has modeled stochastic NUM problems [69], where flows enter and leave the network with time.

We extend this problem by examining the difficulty of designing a network protocol given an *imperfect* model of the network where it will be deployed, in order to understand the inherent difficulties of the problem of congestion control.

In doing so, we draw an explicit analogy to the concept of “learnability” employed in machine learning [58, 50]. A canonical machine-learning problem attempts to design a classifier for a large population of data points, supplied with only a smaller (and possibly skewed) “training set” meant to teach the classifier about the full population of data. Subsequently, the performance of the resulting classifier is evaluated in how well it correctly classifies points in a “testing set,” generally drawn from the actual population. Learnability theory measures the difficulty of inferring an accurate classifier for the testing set, given a training set.

Just as a classifier design procedure may minimize the error rate or the width of the margin as a proxy for maximizing predictive performance on unseen inputs, the Remy tool uses an objective function in terms of throughput and delay, averaged over the design model, as a proxy for performance on as-yet-unseen networks.

In our work, we envision a protocol designer working to generate a congestion-control protocol for a large set of real networks (e.g., the Internet), supplied with only

an imperfect model of the range of variation and behavior of those networks. The imperfect model is the “training scenarios”—a model of the target networks used for design purposes. The “testing scenarios” are drawn from the population of actual target networks.

Learnability here measures the difficulty of designing an adequate protocol for a network *model*, and then deploying it on real networks that cannot be perfectly envisioned at the time of design.

4.3 Experimental setup

We describe our experimental procedure below. First, we specify a set of *training scenarios*: a set of network configurations (§4.3.1) that express the designer’s imperfect model of the network. Next, we specify an *objective function* (§4.3.2). Remy synthesizes a congestion-control protocol that attempts to maximize the value of this function, averaged over the set of training scenarios. Finally (§4.3.3), we specify a *testing scenario* of network configurations, which may be similar or dissimilar to the training scenarios.

We evaluate the synthesized congestion-control protocol on the testing scenario to assess the questions of this study—how easy is it to “learn” a network protocol to achieve desired goals, given an imperfect model of the networks where it will ultimately be deployed?

4.3.1 Training scenarios

The training scenarios specify the set of network configurations that the protocol-design process is given access to. Formally, a network configuration specifies:

1. The topology: link rate and propagation delay of each link in the network.
2. The locations of senders and receivers within the topology, and the paths connecting the senders to the receivers.
3. A model of the workload generated by the application running at each endpoint. We use an on/off model for the workload, where a sender turns “on” for a certain duration (or transfer length) drawn from an exponential distribution, then turns “off” for an amount of time drawn from another exponential distribution before turning on again.
4. The buffer size and queue discipline at each gateway.

4.3.2 Objective function

The objective function expresses the protocol designer’s figure of merit for the goodness of a congestion-control protocol. Many such metrics have been proposed, including alpha-fair throughput [54], flow completion time [3], throughput-over-delay [45], or measures based on a subjective opinion score [24].

In this study, we specifically considered objective functions of the form:

$$\log(\text{throughput}) - \delta \log(\text{delay})$$

Here, “throughput” is the average information transmission rate of a sender-receiver pair, defined as the total number of bytes successfully delivered divided by the total time the sender was “on” and had offered load. The “delay” is the average per-packet delay of packets in the connection, including propagation delay and queueing delay. The δ factor expresses a relative preference between high throughput and low delay.

The protocol-design process works to maximize the *sum* of the objective function across all connections. The log in the objective function expresses a preference for “proportionally-fair” resource allocation—for example, it is worthwhile to cut one connection’s throughput in half, as long as this enables another connection’s throughput to be more-than-doubled.

4.3.3 Evaluation procedure

To measure the difficulty of learning a congestion-control protocol with an imperfect model of the eventual network, we choose a testing scenario of network configurations and evaluate the RemyCC on it. All evaluations are performed in the ns-2 simulator.¹

We compare the performance of RemyCCs optimized for an “accurate” model of the network against RemyCCs optimized for various kinds of imperfect models, in order to measure how faithfully protocol designers need to understand the network they are designing for. We give the training and testing scenarios in a table.

For reference, we also compare the RemyCCs with two common schemes in wide use today:

1. TCP Cubic [23], the default congestion-control protocol in Linux
2. Cubic over stochastic fair queueing and CoDel [44], an active-queue-management scheme that runs on bottleneck routers and assists endpoints in achieving a fairer and more efficient use of network resources

4.4 Measuring the learnability of congestion control

We conducted an experimental study to examine several aspects of congestion-control learnability, by using Remy as an instrument to ask questions of the form: how faithfully do protocol designers really need to understand the networks they design for? What knowledge about the network is important to capture in a design process, and what simplifications are acceptable?

¹Using one simulator for training (Remy’s internal simulator) and a different one for evaluation helps give confidence that the congestion-control protocols learned are robust to quirks in a simulator implementation.

Table 4.1: Training scenarios for “knowledge of network parameters” experiment, showing the effect of varying the intended link rate operating range. Each RemyCC was designed for a network with a single bottleneck, and each sender with a mean “on” and “off” time of 1 s.

RemyCC	Link rates	RTT	Max. number of senders
1000x	1–1000 Mbps	150 ms	2
100x	3.2–320 Mbps	150 ms	2
10x	10–100 Mbps	150 ms	2
2x	22–44 Mbps	150 ms	2

4.4.1 Knowledge of network parameters

We evaluated the difficulty of designing a congestion-control protocol, subject to imperfect knowledge about the parameters of the network.

Some congestion-control protocols have been designed for specific kinds of networks [3, 41] or require explicit knowledge of the link rate *a priori* [32]. Others are intended as one-size-fits-all solutions, including most variants of TCP.

We set out to answer a question posed in [66]: are “one size fits all” protocols inherently at a disadvantage, because of a tradeoff between the “operating range” of a protocol and its performance?

To quantify this, we designed four RemyCCs for training scenarios encompassing a thousand-fold variation in link rates, a hundred-fold variation, a ten-fold variation, and a two-fold variation. Each range was centered on the geometric mean of 1 and 1000 Mbps (32 Mbps), and each set of training scenarios sampled 100 link rates logarithmically from the range. The training scenarios are shown in Table 4.1.

We tested these schemes in ns-2 by sweeping the link speed between 1 and 1000 Mbps, keeping the other details of the simulated network identical to the training scenario. The results are shown Figure 4-1.

We found only weak evidence of a tradeoff between operating range and performance: optimizing for a smaller range did help modestly within that range. Because of the small magnitude of the effect, we cannot say for sure whether simply spending more time optimizing the broader-scope RemyCCs might bring their performance closer to the narrower-scope RemyCCs. Each RemyCC outperformed the human-designed algorithms over its full design range, suggesting that it may be possible to construct “one size fits all” RemyCCs that outperform TCP across a broad range of real-world conditions, even when TCP is assisted by in-network algorithms.

4.4.2 Structural knowledge

We evaluated the difficulty of designing a congestion-control protocol, subject to imperfect knowledge of the network’s structure or topology.

It is an understatement to say that the Internet is a vast network whose full structure is known to nobody and which no model can accurately capture. Nonetheless,

Figure 4-1: Evidence of a weak tradeoff between operating range of a congestion-control protocol and performance. The RemyCC protocols designed with more specific network models (RemyCC-2x and RemyCC-10x) performed modestly better—within their design ranges—than protocols designed for a broader range of networks (RemyCC-100x and RemyCC-1000x), at a cost of deterioration when the actual network did not fall within the training scenarios. The four RemyCC protocols outperformed Cubic and Cubic-over-sfqCoDel over their respective design ranges.

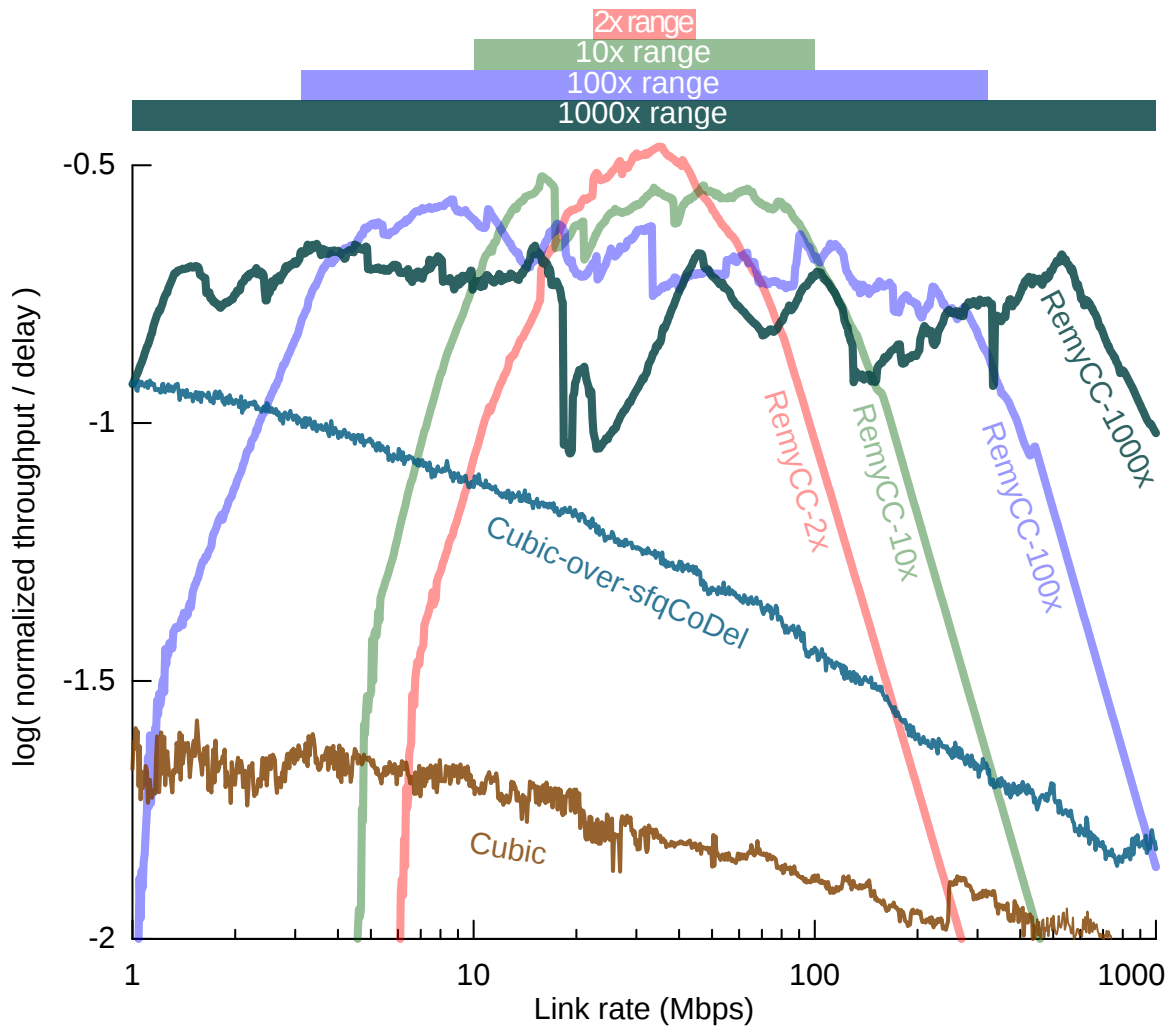
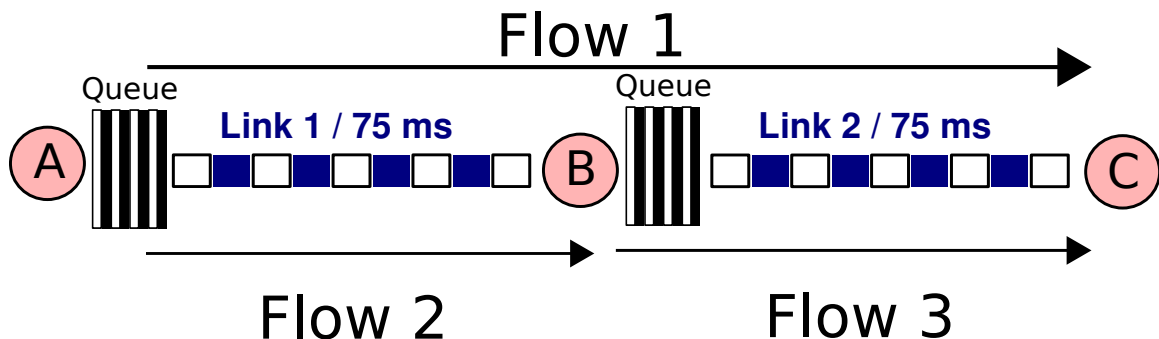


Figure 4-2: Parking-lot topology used to measure the consequences of imperfect knowledge about the network’s structure.



researchers regularly develop new distributed protocols for the Internet, which are deployed based on tests in example network paths that imperfectly capture the Internet’s true complexity.

In practice, protocol designers expect that they can reason about the performance of a distributed network algorithm by modeling the network as something simpler than it is. We worked to capture that intuition rigorously by studying quantitatively how difficult it is to learn a congestion-control protocol for a more-complicated network, given a simplified model of that network’s structure.

In ns-2, we simulated a network with two bottlenecks in a “parking-lot” topology, shown in Figure 4-2. Flow 1 crosses both links and encounters both bottlenecks. It contends with Flow 2 for access to the bottleneck queue at node A, and contends with Flow 3 for access to the bottleneck queue at node B.

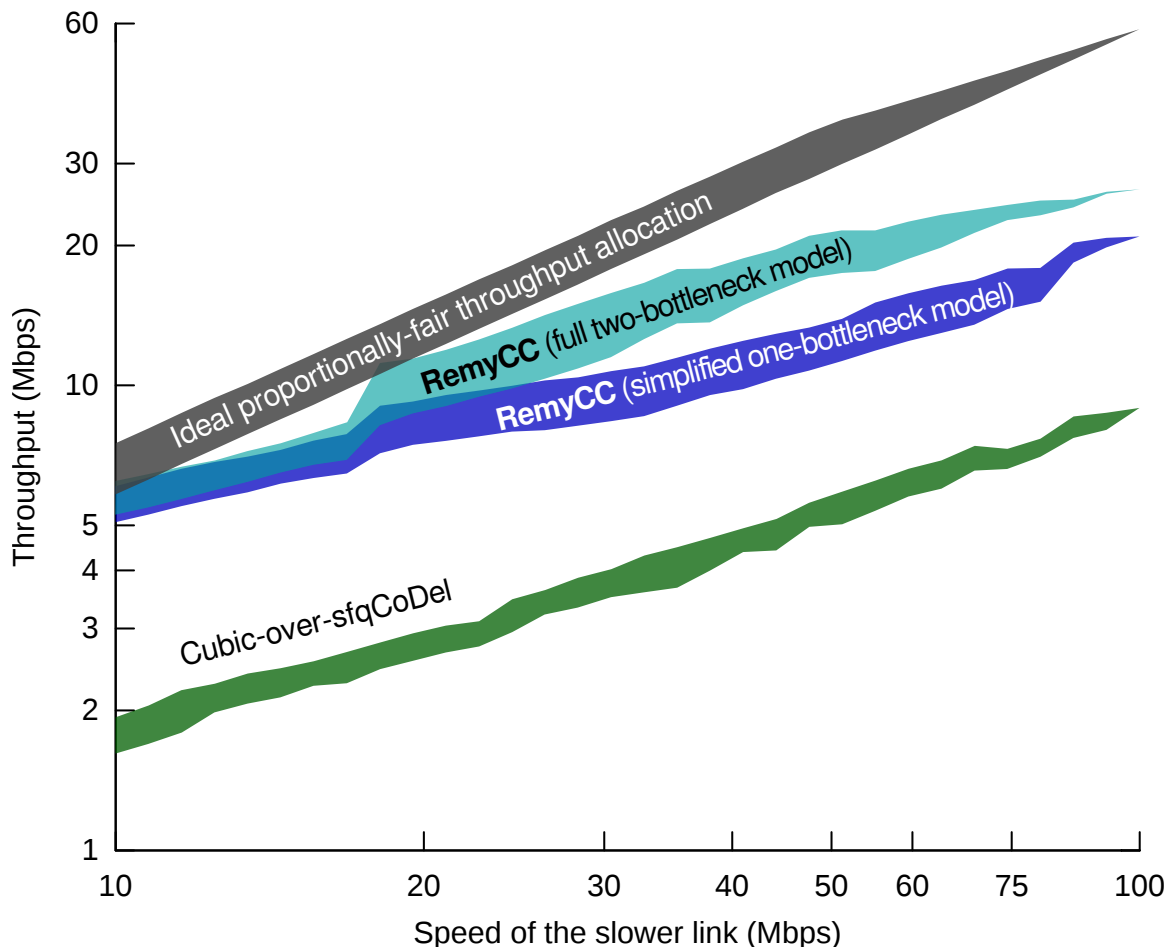
The results for Flow 1 (the flow that crosses both links) are shown in Figure 4-3.² The experiment sweeps the rate of each of the two links between 10 and 100 Mbps, and the shaded area in the figure shows the full locus of throughputs seen by Flow 1. For each pair of rates (for link 1 and link 2), we also calculate the ideal proportionally-fair throughput allocation, and plot the locus of these points as well.

The RemyCC that was designed for the true network achieves close to the ideal proportionally-fair throughput allocation across the two links. The RemyCC that was designed for a simplified model of the network with only one hop also performs adequately, but a little worse than the “true-network” RemyCC. However, both computer-generated protocols come closer to the ideal than contemporary protocols in wide use, TCP Cubic [23], the default TCP in Linux, running over per-flow queueing and CoDel [44], an active queue management (AQM) scheme that runs at both bottlenecks.

The results suggest that a congestion-control protocol designed for a simplified model of the network’s structure can experience a quantifiable—but in this case modest—penalty to its performance.

²Results for Flow 2 and Flow 3 (flows crossing only a single link) were nearly identical between the schemes.

Figure 4-3: How well do endpoints need to understand the network around them? To assess this, we measure the throughput of three congestion-control schemes across a simulated two-hop network path, as the rate of each link is swept between 10 and 100 Mbps, with 75 ms of delay per hop. A RemyCC designed for a simplified one-bottleneck model of the network performs 17% worse, on average, than a RemyCC designed with knowledge of the network’s true two-bottleneck structure. Both RemyCCs outperform TCP Cubic assisted by per-flow queueing and CoDel at the bottlenecks.



4.4.3 Knowledge about other endpoints

We investigated the consequences of designing a congestion-control protocol with the knowledge that some cross-traffic may be the product of pre-existing incumbent protocols.

This question has considerable practical relevance: outside of an internal network controlled by a small number of parties, the developer of a new network protocol will rarely be able to arrange a “flag day” when all endpoints switch to the new protocol.³

³There has not been a “flag day” on the Internet since the switch to IP in 1983.

Table 4.2: Training scenarios used to measure the consequences of imperfect knowledge of the network structure. Both protocols were designed for link rates distributed log-uniformly between 10 and 100 Mbps, and for flows with mean “on” and “off” time of 1 second.

RemyCC	Links modeled	Num. senders
one-bottleneck	one, 150 ms delay	2
true network	two, 75 ms delay each	3

On the broad Internet today, cross-traffic will typically be the product of traditional loss-triggered TCP congestion-control protocols, such as NewReno or Cubic. This status quo presents a problem for new protocols that seek to perform differently or that try to avoid building up standing queues inside the network.

Some protocols, such as Vegas [12], perform well when contending only against other flows of their own kind, but are “squeezed out” by the more-aggressive cross-traffic produced by traditional TCP. Conventional wisdom is that any “delay-based” protocol will meet a similar fate. This has contributed to a lack of adoption of Vegas and other delay-based protocols.

Ideally, a newly-designed protocol would perform well (e.g. high throughput, low delay) when interacting with other endpoints running the same protocol, **and** would appropriately share a network with incumbent endpoints running traditional TCP. But what are the consequences of building this “TCP-awareness” into a protocol?

We studied this by designing two network protocols for a simple network—one whose model specified that the cross-traffic would be from the same protocol, and one whose model included a training scenario where the cross-traffic was from traditional TCP half the time.

In Figure 4-4, protocols compete only with cross-traffic from the same protocol. In this homogeneous setting, adding TCP-awareness to a RemyCC builds up standing queues, more than doubling the queueing delay without affecting throughput.

But in a mixed setting (Figure 4-5) where a RemyCC competes against TCP NewReno, the TCP-naive RemyCC is squeezed out and does not get its fair share of the link. In the shaded region showing the results when NewReno competes with the TCP-aware RemyCC, TCP-awareness allows the RemyCC to claim its fair share of the link and reduces the queueing delay experienced both by itself and by TCP NewReno.

The results indicate that new delay-minded protocols *can* avoid being squeezed out by traditional loss-triggered TCP, but building in this behavior comes at a cost to performance in the absence of TCP cross-traffic.

Figure 4-4: Performance of two flows of the same congestion-control scheme contending for a 10 Mbps link, for three different congestion-control schemes. The “TCP-naive” RemyCC is designed under the assumption that its competition, when present, is governed by the same RemyCC—an assumption that holds here. The “TCP-aware” RemyCC is designed under the assumption that its cross traffic has a 50% chance of being governed by a TCP AIMD-like scheme. The results quantify the degree to which designing a protocol to play well with TCP can exact a cost when TCP is absent.

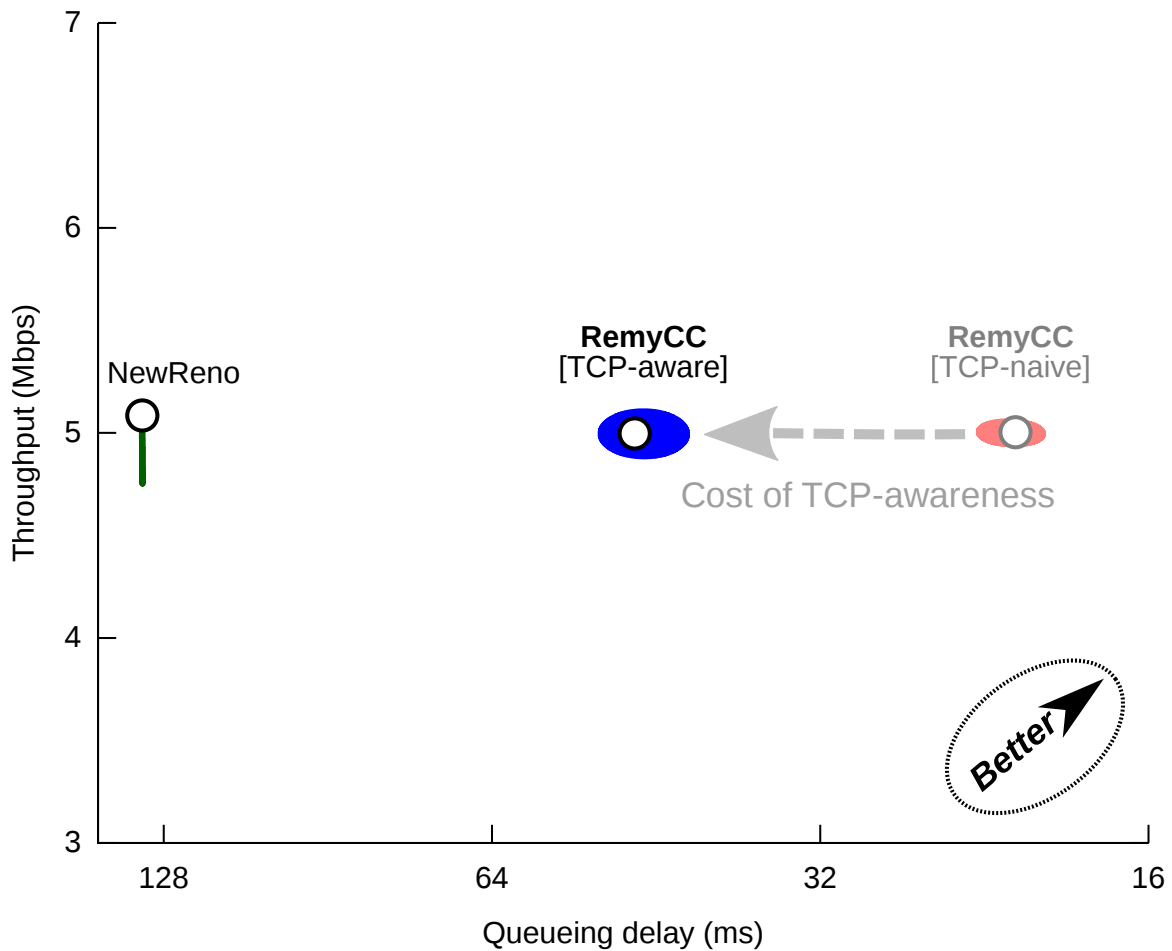
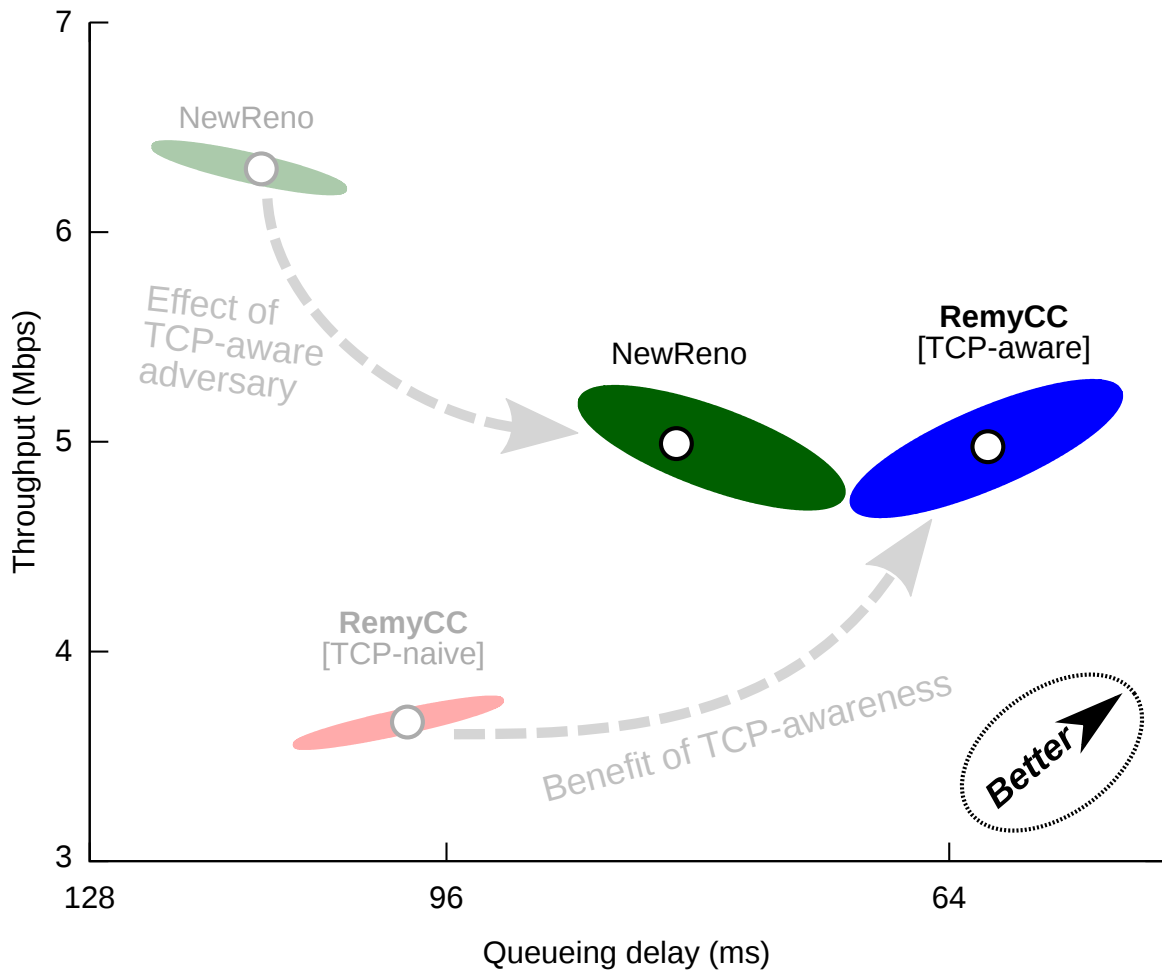


Figure 4-5: When cross traffic is governed by TCP NewReno, the TCP-aware RemyCC achieves a more equitable division of the link—and better delay for both flows—than the TCP-naïve RemyCC. The results demonstrate the benefit of TCP-awareness when TCP is present.



Chapter 5

Discussion

The practical impact of Sprout and Remy so far remains preliminary. There have been several experiments on Sprout conducted by companies interested in using it as a control algorithm for videoconferencing or screen-sharing software. Generally speaking, I understand these experiments have confirmed the ones in this thesis, under the assumption that Sprout’s request for bytes from the application can always be satisfied.

However, in practice, coupling Sprout to an actual video coder has presented several new challenges. Real video coders generally do not have the agility to produce frames of exactly the requested coded size, immediately on demand. Today’s videoconferencing applications typically allow the video coder to “drive” the application, coding and sending video at a particular nominal bit rate until the transport receives definitive evidence of congestion. Flipping that around—so that Sprout’s stochastic forecasts can “drive” the video coder—may require changes to the video coder to be able to vary the coded size of frames more rapidly.

As for Remy, much remains unknown about the capabilities and limits of computer-generated algorithms, much less decentralized algorithms that cooperate indirectly across a network to achieve a common goal. Although the RemyCCs appear to work well on networks whose parameters fall within or near the limits of what they were prepared for—even beating in-network schemes at their own game and even when the design range spans three orders of magnitude—our ability to reason about *why* they work is limited.

We can observe apparent limit cycles with the Ratatouille tool in simulation and make some statements about the behavior of RemyCCs on new kinds of simulated networks, but we do not have a principled method to predict a RemyCC’s performance on real-world heterogeneous networks with unknown link parameters, topology, and cross-traffic. Indeed, we have yet to run RemyCCs on real-world networks at all.

Along with the rigor of the computer-generated approach come new challenges for protocol designers, who must figure out how to encode their intuitions about the problem into an explicit statement of assumptions and an objective. How should a protocol designer come up with their model of the uncertain network? Our learnability experiments in Chapter 4 suggest that, even on networks with more-complex structure, it may be sufficient for a RemyCC to have been designed for the link rate and degree of

multiplexing of its *bottleneck* link only. Much work needs to be done, however, before we can reliably extrapolate from results on a two-bottleneck parking-lot topology to more-complex networks. It may also be possible for a RemyCC to “learn” different congestion policies for different remote IP addresses, but the cross-interaction between such adaptive RemyCCs—who might be executing different policies at runtime—will need to be studied.

It was a surprising result that computer-generated end-to-end RemyCCs were able to outperform some of the most sophisticated human-designed in-network schemes, including Cubic-over-sfqCoDel and XCP. On the one hand, this result is encouraging—it indicates that end-to-end algorithms are more powerful than was previously understood, perhaps making it possible to preserve a “dumb” network with smart endpoints.

In general, though, an in-network algorithm that can run on the bottleneck gateway and enforce fairness among flows (as sfqCoDel and XCP both do) is strictly more powerful than a decentralized algorithm that runs purely end-to-end. My colleagues and I have argued [52] that, like endpoint congestion-control schemes, in-network algorithms will need to keep evolving with time. I expect that if similar methods to Remy are applied to the development of in-network algorithms, they will be able to reclaim their superiority over the best end-to-end algorithms.

Bibliography

- [1] A. Aggarwal, S. Savage, and T. Anderson. Understanding the performance of TCP pacing. In *INFOCOM 2000*, volume 3, pages 1157–1165. IEEE, 2000.
- [2] A. Akella, S. Seshan, R. Karp, S. Shenker, and C. Papadimitriou. Selfish Behavior and Stability of the Internet: A Game-Theoretic Analysis of TCP. In *SIGCOMM*, 2002.
- [3] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data Center TCP (DCTCP). In *SIGCOMM*, 2010.
- [4] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker. pFabric: Minimal near-optimal datacenter transport. In *SIGCOMM*, pages 435–446. ACM, 2013.
- [5] M. Allman. Initial Congestion Window Specification. <http://tools.ietf.org/html/draft-allman-tcpm-bump-initcwnd-00>, 2010.
- [6] M. Allman. Comments on Bufferbloat. *SIGCOMM Computer Communication Review*, 43(1), Jan. 2013.
- [7] H. Balakrishnan, H. S. Rahul, and S. Seshan. An Integrated Congestion Management Architecture for Internet Hosts. In *SIGCOMM*, 1999.
- [8] D. Bansal and H. Balakrishnan. Binomial Congestion Control Algorithms. In *INFOCOM*, 2001.
- [9] D. Bansal, H. Balakrishnan, S. Floyd, and S. Shenker. Dynamic Behavior of Slowly-Responsive Congestion Control Algorithms. In *SIGCOMM*, 2001.
- [10] P. Bender, P. Black, M. Grob, R. Padovani, N. Sindhuhyana, and A. Viterbi. A bandwidth efficient high speed wireless data service for nomadic users. *IEEE Communications Magazine*, July 2000.
- [11] D. S. Bernstein, R. Givan, N. Immerman, and S. Zilberstein. The Complexity of Decentralized Control of Markov Decision Processes. *Mathematics of Operations Research*, 27(4):819–840, Nov. 2002.

- [12] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson. TCP Vegas: New Techniques for Congestion Detection and Avoidance. In *SIGCOMM*, 1994.
- [13] D.-M. Chiu and R. Jain. Analysis of the Increase and Decrease Algorithms for Congestion Avoidance in Computer Networks. *Computer Networks and ISDN Systems*, 17:1–14, 1989.
- [14] J. Chu, N. Dukkupati, Y. Cheng, and M. Mathis. Increasing TCP's Initial Window. <http://tools.ietf.org/html/draft-ietf-tcpm-initcwnd-08>, 2013.
- [15] D. Clark. The Design Philosophy of the DARPA Internet Protocols. In *SIGCOMM*, 1988.
- [16] D. Cox. Long-range dependence: A review. In *H.A. David and H.T. David, editors, Statistics: An Appraisal*, pages 55–74. Iowa State University Press, 1984.
- [17] N. Dukkupati, T. Refice, Y. Cheng, J. Chu, T. Herbert, A. Agarwal, A. Jain, and N. Sutin. An Argument for Increasing TCP's Initial Congestion Window. *SIGCOMM Computer Communication Review*, 40(3):27–33, 2010.
- [18] W. Feng, K. Shin, D. Kandlur, and D. Saha. The BLUE Active Queue Management Algorithms. *IEEE/ACM Transactions on Networking*, Aug. 2002.
- [19] S. Floyd. TCP and Explicit Congestion Notification. *CCR*, 24(5), Oct. 1994.
- [20] S. Floyd, M. Handley, J. Padhye, and J. Widmer. Equation-Based Congestion Control for Unicast Applications. In *SIGCOMM*, 2000.
- [21] S. Floyd and V. Jacobson. Random Early Detection Gateways for Congestion Avoidance. *IEEE/ACM Transactions on Networking*, 1(4), Aug. 1993.
- [22] J. Gettys and K. Nichols. Bufferbloat: Dark buffers in the internet. *Queue*, 9(11):40:40–40:54, Nov. 2011.
- [23] S. Ha, I. Rhee, and L. Xu. CUBIC: A New TCP-Friendly High-Speed TCP Variant. *ACM SIGOPS Operating System Review*, 42(5):64–74, July 2008.
- [24] O. Habachi, Y. Hu, M. van der Schaar, Y. Hayel, and F. Wu. MOS-based congestion control for conversational services in wireless environments. *IEEE Journal on Selected Areas in Communications*, 30(7):1225–1236, August 2012.
- [25] J. C. Hoe. Improving the Start-up Behavior of a Congestion Control Scheme for TCP. In *SIGCOMM*, 1996.
- [26] D. Hofstadter. *Metamagical Themas: Questing for the Essence of Mind and Pattern*. Basic books, 1985.
- [27] T.-Y. Huang, N. Handigol, B. Heller, N. McKeown, and R. Johari. Confused, timid, and unstable: picking a video streaming rate is hard. In *Proceedings of the 2012 ACM conference on Internet measurement conference*, pages 225–238. ACM, 2012.

- [28] V. Jacobson. Congestion Avoidance and Control. In *SIGCOMM*, 1988.
- [29] R. Jain. A Delay-based Approach for Congestion Avoidance in Interconnected Heterogeneous Computer Networks. In *SIGCOMM*, 1989.
- [30] H. Jiang, Y. Wang, K. Lee, and I. Rhee. Tackling bufferbloat in 3G/4G mobile networks. *NCSU Technical report*, March 2012.
- [31] P. Karn, C. Bormann, G. Fairhurst, D. Grossman, R. Ludwig, J. Mahdavi, G. Montenegro, J. Touch, and L. Wood. Advice for Internet Subnetwork Designers, 2004. RFC 3819, IETF.
- [32] D. Katabi, M. Handley, and C. Rohrs. Congestion Control for High Bandwidth-Delay Product Networks. In *SIGCOMM*, 2002.
- [33] F. P. Kelly, A. Maulloo, and D. Tan. Rate Control in Communication Networks: Shadow Prices, Proportional Fairness and Stability. *Journal of the Operational Research Society*, 49:237–252, 1998.
- [34] S. Keshav. Packet-Pair Flow Control. *IEEE/ACM Transactions on Networking*, Feb. 1995.
- [35] E. Kohler, M. Handley, and S. Floyd. Designing DCCP: Congestion control Without Reliability. In *SIGCOMM*, 2006.
- [36] S. Kunniyur and R. Srikant. Analysis and Design of an Adaptive Virtual Queue (AVQ) Algorithm for Active Queue Management. In *SIGCOMM*, 2001.
- [37] T. Lan, D. Kao, M. Chiang, and A. Sabharwal. An Axiomatic Theory of Fairness. In *INFOCOM*, 2010.
- [38] D. Leith and R. Shorten. H-TCP Protocol for High-Speed Long Distance Networks. In *PFLDNet*, 2004.
- [39] H. Lundin, S. Holmer, and H. Alvestrand. A Google Congestion Control Algorithm for Real-Time Communication on the World Wide Web. <http://tools.ietf.org/html/draft-alvestrand-rtcweb-congestion-03>, 2012.
- [40] R. Mahajan, J. Padhye, S. Agarwal, and B. Zill. High performance vehicular connectivity with opportunistic erasure coding. In *USENIX*, 2012.
- [41] S. Mascolo, C. Casetti, M. Gerla, M. Sanadidi, and R. Wang. TCP Westwood: Bandwidth Estimation for Enhanced Transport over Wireless Links. In *MobiCom*, 2001.
- [42] P. E. McKenney. Stochastic Fairness Queueing. In *INFOCOM*, 1990.
- [43] D. Meagher. Geometric Modeling Using Octree Encoding. *Computer Graphics and Image Processing*, 19(2):129–147, 1982.

- [44] K. Nichols and V. Jacobson. Controlling Queue Delay. *ACM Queue*, 10(5), May 2012.
- [45] K. Nichols and V. Jacobson. Controlled Delay Active Queue Management. <http://tools.ietf.org/html/draft-nichols-tsvwg-codel-02>, 2014.
- [46] F. A. Oliehoek. Decentralized POMDPs. In *Reinforcement Learning: State of the Art, Adaptation, Learning, and Optimization*, pages 471–503, 2012.
- [47] R. Pan, B. Prabhakar, and K. Psounis. CHOKe—A Stateless Active Queue Management Scheme for Approximating Fair Bandwidth Allocation. In *INFOCOM*, 2000.
- [48] V. Paxson and S. Floyd. Wide-Area Traffic: The Failure of Poisson Modeling. *IEEE/ACM Transactions on Networking*, 3(3):226–244, June 1995.
- [49] K. K. Ramakrishnan and R. Jain. A Binary Feedback Scheme for Congestion Avoidance in Computer Networks. *ACM Trans. on Comp. Sys.*, 8(2):158–181, May 1990.
- [50] R. E. Schapire. The strength of weak learnability. *Machine learning*, 5(2):197–227, 1990.
- [51] S. Shalunov, G. Hazel, J. Iyengar, and M. Kuehlewind. Low Extra Delay Background Transport (LEDBAT), 2012. IETF RFC 6817.
- [52] A. Sivaraman, K. Winstein, S. Subramanian, and H. Balakrishnan. No silver bullet: extending SDN to the data plane. In *HotNets-XII*, 2013.
- [53] A. Sivaraman, K. Winstein, P. Varley, S. Das, J. Ma, A. Goyal, J. Batalha, and H. Balakrishnan. Protocol design contests. *SIGCOMM Computer Communication Review*, July 2014 (forthcoming).
- [54] R. Srikant. *The Mathematics of Internet Congestion Control*. Birkhauser, 2004.
- [55] C. Tai, J. Zhu, and N. Dukkipati. Making Large Scale Deployment of RCP Practical for Real Networks. In *INFOCOM*, 2008.
- [56] K. Tan, J. Song, Q. Zhang, and M. Sridharan. A Compound TCP Approach for High-speed and Long Distance Networks. In *INFOCOM*, 2006.
- [57] J. Touch. Automating the Initial Window in TCP. <http://tools.ietf.org/html/draft-touch-tcpm-automatic-iw-03>, 2012.
- [58] L. G. Valiant. A Theory of the Learnable. *CACM*, 27(11):1134–1142, Nov. 1984.
- [59] A. Venkataramani, R. Kokku, and M. Dahlin. TCP Nice: a mechanism for background transfers. *SIGOPS Oper. Syst. Rev.*, 36(SI):329–343, Dec. 2002.

- [60] Z. Wang and J. Crowcroft. A New Congestion Control Scheme: Slow Start and Search (Tri-S). In *SIGCOMM*, 1991.
- [61] D. Wei, C. Jin, S. Low, and S. Hegde. FAST TCP: Motivation, Architecture, Algorithms, Performance. *IEEE/ACM Transactions on Networking*, 14(6):1246–1259, 2006.
- [62] J. Widmer, R. Denda, and M. Mauve. A survey on TCP-friendly congestion control. *IEEE Network*, 15(3):28–37, 2001.
- [63] K. Winstein and H. Balakrishnan. End-to-End Transmission Control by Modeling Uncertainty about the Network State . In *HotNets-X*, 2011.
- [64] K. Winstein and H. Balakrishnan. TCP ex Machina: Computer-Generated Congestion Control. In *SIGCOMM*, Hong Kong, China, August 2013.
- [65] K. Winstein, A. Sivaraman, and H. Balakrishnan. Stochastic Forecasts Achieve High Throughput and Low Delay over Cellular Networks. In *NSDI*, Lombard, Ill., April 2013.
- [66] J. Wroclawski. TCP ex Machina. <http://www.postel.org/pipermail/end2end-interest/2013-July/008914.html>, 2013.
- [67] Y. Xia, L. Subramanian, I. Stoica, and S. Kalyanaraman. One More Bit is Enough. *IEEE/ACM Transactions on Networking*, 16(6):1281–1294, 2008.
- [68] L. Xu, K. Harfoush, and I. Rhee. Binary Increase Congestion Control (BIC) for Fast Long-Distance Networks. In *INFOCOM*, 2004.
- [69] Y. Yi and M. Chiang. Stochastic Network Utility Maximisation. *European Transactions on Telecommunications*, 19(4):421–442, 2008.